TURING 图灵程序设计丛书

SAMS

Sams Teach Yourself Regular Expressions in 10 Minutes

正则表达式必知必会

[美] Ben Forta 著

杨涛 等译



- 正则表达式经典著作
- 轻松迅捷的学习环境
- 示例丰富，涵盖所有主流平台



人民邮电出版社

POSTS & TELECOM PRESS

Sams Teach Yourself Regular Expressions in 10 Minutes

正则表达式必知必会

"如果你希望能够迅速掌握正则表达式，我极力推荐此书。"

——www.regular-expressions.info

"简明扼要、轻松易读，非常优秀的入门书。"

——Amazon.com 评语

正则表达式是一种威力无比强大的武器，可以完成各种复杂的文本处理工作，被称为程序员的"瑞士军刀"。

本书是全球程序员公认的正则表达式最佳速成教材。与动辄几百页的同类书相比，本书的最大优势是精选正则表达式中最常用、最重要的知识，通过简明又实用的示例，从简单的文本匹配开始，循序渐进地介绍更高级的内容，包括回溯引用、前后查找、嵌入条件，等等。而且书中的内容在保持语言和平台中立的同时，还兼顾了各种平台之间的差异。通过阅读本书，读者能够在轻松的氛围中迅速掌握正则表达式的精髓，并可立即运用所学，解决实际问题。



Ben Forta 是世界知名的技术作家，也是Adobe技术界最为知名的专家之一，目前担任Adobe公司的高级技术推广专家。他具有计算机行业20多年工作经验，多年来撰写了十几本技术图书，其中不少是世界畅销书，已被翻译为十几种文字。除本书外，他撰写的《SQL必知必会》也由人民邮电出版社出版。读者可以通过他的个人网站 http://www.forta.com 了解更多信息。

SAMS

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)88593802
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/程序设计

人民邮电出版社网址 www.ptpress.com.cn



ISBN 978-7-115-16474-2
9 787115 164742 >
ISBN 978-7-115-16474-2/TP
定价：29.00 元

TURING 图灵程序设计丛书

正则表达式必知必会

Sams Teach Yourself Regular Expressions in 10 Minutes

[美] Ben Forta 著
杨涛 等译

人民邮电出版社
北 京

图书在版编目（CIP）数据

正则表达式必知必会 / （美）福达（Forta,B.）著；
杨涛等译. —北京：人民邮电出版社，2007.12
（图灵程序设计丛书）
ISBN 978-7-115-16474-2

Ⅰ. 正… Ⅱ. ①福…②杨… Ⅲ. 正则表达式—教材
Ⅳ. TP301.2

中国版本图书馆 CIP 数据核字（2007）第 096517 号

内 容 提 要

正则表达式是一种威力无比强大的武器，几乎在所有的程序设计语言里和计算机平台上都可以用它来完成各种复杂的文本处理工作。本书从简单的文本匹配开始，循序渐进地介绍了很多复杂内容，其中包括回溯引用、条件性求值和前后查找，等等。每章都为读者准备了许多简明又实用的示例，有助于全面、系统、快速掌握正则表达式，并运用它们去解决实际问题。

本书适合各种语言和平台的开发人员。

图灵程序设计丛书

正则表达式必知必会

◆ 著　　[美] Ben Forta
译　　杨 涛 等
责任编辑　陈兴璐

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800×1230　1/32
印张：4.75
字数：146 千字　　2007 年 12 月第 1 版
印数：1 ~ 5 000 册　　2007 年 12 月北京第 1 次印刷
著作权合同登记号　图字：01-2007-1961 号

ISBN 978-7-115-16474-2/TP

定价：29.00 元

读者服务热线：(010)88593802　印装质量热线：(010)67129223

版 权 声 明



引　言

正则表达式（regular expression）和正则表达式语言已经出现很多年了。正则表达式的专家们早就掌握了这种威力无比强大的武器，它可以用来完成各种复杂的文本处理工作。更重要的是，这种武器可以在几乎所有的程序设计语言里和几乎所有的计算机平台上使用。

这是个好消息，但我还要告诉你一个坏消息：长期以来，只有一些真正的专家才能真正掌握正则表达式。甚至有很多人根本没有听说过正则表达式这个概念，更不用说用它们来解决问题了。至于少数勇于涉猎正则表达式领域的人们，又往往会因为正则表达式难以理解而浅尝辄止或总是在原地徘徊。这不能不说是一种悲哀，因为正则表达式其实并没有人们想像中的那么复杂。只要你能清晰地理解你想要解决的问题并学会如何使用正则表达式，就可以轻而易举地解决这些问题。

正则表达式不为大多数人所掌握的原因之一是关于这方面的好资料太少了。虽然有很多网站在吹嘘它们的正则表达式教程如何全面，但实际情况却是高质量的正则表达式学习资源相当稀缺。即便能够找到几本介绍正则表达式的书籍，它们又往往过于偏重语法而显得不够实用——知道如何定义{或是知道+与*之间的区别并不等于真正掌握了正则表达式的用法。在笔者看来，那些书籍反而把简单的问题弄得更复杂了：在学习和使用正则表达式的时候，重要的并不是你知道多少个特殊字符，而是你会不会运用它们去解决实际问题。

你拿在手里的这本书并不打算成为一本正则表达式的大全。如果你想要的是那样一本书，你应该去阅读Jeffrey Friedl编写的*Mastering Regular Expressions*（O'Reilly出版公司，ISBN 0596002890）。Friedl先生是业内公认的正则表达式专家，他的书绝对是这方面最权威和全面的著

作。本人对Friedl先生没有丝毫成见，但他的书不适合初学者也是实情；如果你只打算尽快完成手头的工作而不是要钻研正则表达式的内部原理的话，他的书也不很适用。这并不是说那本书里的信息没有用，只是它在你想要给HTML表单添加一些验证功能或者只想对解析的文本进行替换的时候派不上什么用场。如果你想尽快学会正则表达式的基本用法，你将发现自己陷入了一个两难境地：要么找不到简明易学的参考资料，要么找到的参考资料过于深奥而让你不知该如何起步。

这正是促使笔者编写本书的原因。本书所讲授的关于正则表达式知识正是你们在刚起步时最需要的，我们将从简单的文本匹配开始循序渐进地向大家介绍许多复杂的专题，其中包括回溯引用（backreference，或译为后向引用）、条件性求值（conditional evaluation）和前后查找（looking-around），等等。本书最大的优势是所学到的知识可以立即运用于实践中：我们在每章里都为大家准备了许多简明又实用的示例，它们可以帮助你全面、系统、快速地掌握正则表达式并运用它们去解决实际问题，而每章在10分钟甚至更短的时间里就可以学完。

还等什么，赶快翻到第1章开始今天的学习吧，你肯定会立刻感受到正则表达式的强大威力。

目标读者

本书的目标读者是以下几类人员：

- 第一次接触正则表达式。
- 希望自己能够快速掌握正则表达式的基本用法。
- 想使用一种强大的工具（虽然它不那么容易掌握）去解决实际问题。
- 正在开发Web应用软件并需要进行复杂的表单和文本处理。
- 正使用着Perl、ASP、Visual Basic、.NET、C#、Java、JSP、PHP、ColdFusion语言（或更多其他程序设计语言），希望在开发的应用程序里使用正则表达式。
- 希望在不求助于其他人的前提下尽快掌握正则表达式。

致谢

首先，我要感谢正则表达式专家和我以前的合作者Michael Dinowitz，他对本书的技术细节进行了严格的审校并提供了许多宝贵的意见和反馈。

本书的附录C向大家介绍了一种基于Web的正则表达式测试器，而我必须在此感谢这个测试器的原始作者Nate Weiss（它最初是为*ColdFusion Web Application Construction Kit*一书而编写的）。在Nate的许可和支持下，我对他用ColdFusion编写的正则表达式测试软件进行了改写以配合本书使用，开发了相应的JavaScript版本。感谢Qasim Rasheed为这个测试器编写ASP和JSP版本，感谢Scott Van Vliet为这个测试器编写ASP.NET版本。

最后，我还要感谢Sams出版公司里帮助我把本书从概念变成现实的人们，尤其是Michael Stephens和Mark Renfrow。没有他们的帮助和支持，本书是不可能与大家见面的。

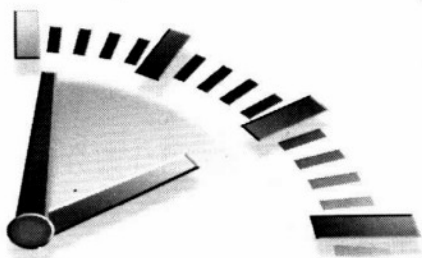
谢谢大家。

——Ben Forta



目　录





第1章 正则表达式入门

在本章里，你将学习何为正则表达式以及它们可以帮助你做些什么。

1.1 正则表达式的用途

正则表达式（regular expression，简称regex）是一种工具，和其他工具一样，它是人们为了解决某一类专门的问题而发明的。要想理解正则表达式及其功用，最好的办法是了解它们可以解决什么样的问题。

请考虑以下几个场景：

- 你正在搜索一个文件，这个文件里包含着单词car（不区分字母大小写），但你并不想把包含着字符串car的其他单词（比如scar、carry和incarcerate，等等）也找出来。
- 你打算用一种应用服务器来动态地生成一个Web网页以显示从某个数据库里检索出来的文本。在那些文本里可能包含着一些URL地址字符串，而你希望那些URL地址在最终生成的页面里是可点击的（也就是说，你打算生成一些合法的HTML代码——<A HREF></A>——而不仅仅是普通的文本）。
- 你创建了一份包含着一张表单的Web页面，这张表单用来收集用户信息，其中包括一个电子邮件地址。你需要检查用户给出的电子邮件地址是否符合正确的语法格式。
- 你正在编辑一段源代码并且要把所有的size都替换为isize，但这种替换仅限于单词size本身而不涉及那些包含着字符串size的其他单词。

- 你正在显示一份计算机文件系统中所有文件的清单，但你只想把文件名里包含着Application字样的文件列举出来。
- 你正在把一些数据导入应用程序。那些数据以制表符作为分隔符，但你的应用程序要支持CSV格式（每条记录独占一行，同一条记录里的各项数据之间用逗号分隔并允许被括在引号里面）。
- 你需要在文件里搜索某个特定的文本，但你只想把出现在特定位置的（比如每行的开头或是每条语句的结尾）找出来。

以上场景都是大家在编写程序时经常会遇到的问题，用任何一种支持条件处理和字符串操作的编程语言都可以解决它们——但问题是你的解决方案将会变得十分复杂。比较容易想到的办法是，用一些循环来依次遍历那些单词或字符并在循环体里面用一系列 if 语句来进行测试，这往往意味着你需要使用大量的标志来标记你已经找到了什么，你还没有找到什么，还需要检查空白字符和特殊字符，等等。而这一切都需要以手工方式来进行。

另一种解决方案是使用正则表达式。上述问题都可以用一些精心构造的语句——或者说一些由文本和特殊指令构成的高度简练的字符串来解决，比如像下面这样的语句：

```
\b[Cc][Aa][Rr]\b
```

4



注意　如果你现在还看不懂这一行，先别着急。你很快就会知道它的含义是什么。

1.2　如何使用正则表达式

如果认真思考一下那些问题场景，你就会发现它们不外乎两种情况：一种是查找特定的信息（搜索），另一种是查找并编辑特定的信息（替换）。事实上，从根本上来讲，那正是正则表达式的两种基本用途：搜索和替换。给定一个正则表达式，它要么匹配一些文本（进行一次搜索），要么匹配并替换一些文本（进行一次替换）。

1.2.1 用正则表达式进行搜索

正则表达式的主要用途之一是搜索变化多端的文本，比如刚才描述的搜索单词car的场景：你要把car、CAR、Car，或CaR都找出来，但这只是整个问题比较简单的一部分（有许多搜索工具都可以完成不区分字母大小写的搜索）。比较困难的部分是确保scar、carry和incarcerate之类的单词不会被匹配到。一些比较高级的编辑器提供了“Match Only Whole Word（仅匹配整个单词）”选项，但还有许多编辑器并不具备这一功能，而你往往无法在你正在编辑的文档里做出这种调整。使用正则表达式而不是纯文本car进行搜索就可以解决这个问题。



提示 想知道如何解决这个问题吗？你们其实已经见过答案了——它就是我们刚才给出的示例语句：\b[Cc][Aa][Rr]\b

请注意，“等于”比较（比如说，用户给出的电子邮件地址是否匹配这个正则表达式？）本质上也是一种搜索操作，这种搜索操作会对用户所提供的整个字符串进行搜索以寻找一个匹配。与此相对的是子字符串搜索，子字符串搜索是“搜索”这个词的普通含义。

1.2.2 用正则表达式进行替换

正则表达式搜索的威力非常强大，非常有用，而且比较容易学习和掌握。本书的许多章节和示例都与“匹配”有关。不过，正则表达式的真正威力体现在替换操作方面，比如我们刚才所描述的需要把URL地址字符串替换为可点击URL地址的场景：这需要先把相关文本里的URL地址字符串找出来（比如说，通过搜索以http://或https://开头、以句号、逗号或空白字符结尾的字符串），再把找到的URL地址字符串替换为HTML语言的“<A HREF=…> … </A>”元素，如下所示：

5

```
http://www.forta.com/
```

替换结果：

```
<A HREF="http://www.forta.com">http://www.forta.com/</A>
```

绝大多数应用程序的“Search and Replace”（搜索和替换）选项都可

以完成这种替换操作，但使用一个正则表达式来完成这个任务将简单得让人难以置信。

1.3　什么是正则表达式

现在，你已经知道正则表达式是用来干什么的了，我们再来给它下个定义。简单地说，正则表达式是一些用来匹配和处理文本的字符串。正则表达式是用正则表达式语言创建的，这种语言的用途就是为了解决我们前面所描述的种种问题。与其他程序设计语言一样，正则表达式语言也有需要你们去学习的特殊语法和指令，它们正是本书要教给大家的东西。

正则表达式语言并不是一种完备的程序设计语言，它甚至算不上是一种能够直接安装并运行的程序。更准确地说，正则表达式语言是内置于其他语言或软件产品里的“迷你”语言。好在现在几乎所有的语言或工具都支持正则表达式，但是正则表达式与你正在使用的语言或工具可以说毫无相似之处。正则表达式语言虽然也被称为一种语言，但它与人们对语言的印象相去甚远。

6



注意　正则表达式起源于1950年代在数学领域的一些研究工作。几年之后，计算机领域借鉴那些研究工作的成果和思路开发出了Unix世界里的Perl语言和grep等工具程序。在许多年里，正则表达式只流行于Unix平台（Unix程序员用它们来解决我们前面所描述的各种问题），但这种情况早已发生了变化，现在几乎所有的计算平台都支持正则表达式，只是具体方式和支持程度略有差异而已。

说完这些掌故，我们再来看几个例子。下面都是合法的正则表达式（我们稍后再解释它们的用途）：

- Ben
- .
- www\ .forta\ .com

- [a-zA-Z0-9_.]*
- <[Hh]1>.*</[Hh]1>
- \r\n\r\n
- \d{3,3}-\d{3,3}-\d{4,4}

请注意，语法是正则表达式最容易掌握的部分，真正的挑战是学会如何运用那些语法把实际问题分解为一系列正则表达式并最终解决。与学习其他程序设计语言一样，只靠读书是学不会如何灵活运用语法正则的，你必须通过亲身实践才能真正掌握它们。

1.4 使用正则表达式

7

正如前面解释的那样，不存在所谓的正则表达式程序；它既不是可以直接运行的应用程序，也不是可以从哪里购买或下载来的软件。在绝大多数的软件产品、编程语言、工具程序和开发环境里，正则表达式语言都已被实现。

正则表达式的使用方法和具体功能，在不同的应用程序/语言中各有不同。一般来说，应用程序大多使用菜单选项和对话框来访问正则表达式，而程序设计语言大都在函数或对象类中使用正则表达式。

此外，并非所有的正则表达式实现都是一样的。在不同的应用程序/语言里，正则表达式的语法和功能往往会有明显（有时也不那么明显）的差异。

附录A对支持正则表达式的许多应用程序和语言在这方面的细节进行了汇总。在继续学习下一章之前，你应该先熟悉一下附录A，看看你们正在使用的应用程序或语言在正则表达式方面都有哪些与众不同之处。

为了帮助大家尽快入门，我们在这本书的配套网页http：//www.forta.com/books/0672325667/上准备了一个名为“Regular Expression Tester（正则表达式测试器）”的工具软件供大家下载。这个基于Web的工具软件有好几种版本，它们分别对应着一些比较流行的应用服务器和编程语言，还有一个版本是专门用来直接测试用JavaScript语言编写出来的正则表达

式的。附录C对这个工具软件的用法进行了介绍，这个工具可以简便、快速地对你们构造出来的正则表达式进行测试，这对大家的学习肯定会有很大的帮助。

1.5　在继续学习之前

在继续学习之前，你还应该了解以下几个事实：

- 在使用正则表达式的时候，你将发现几乎所有的问题都有不止一种解决方案。它们有的比较简单，有的比较快速，有的兼容性更好，有的功能更全。这么说吧，在编写正则表达式的时候，只有对、错两种选择的情况是相当少见的——同一个问题往往会有多种解决方案。
- 正如我们前面讲过的那样，正则表达式的不同实现往往会有所差异。在编写本书的时候，我们已尽了最大努力来保证各章里的示例能适用于尽可能多的实现；但有些差异和不兼容是无法回避的，我们针对这种情况都尽可能地进行了注明。
- 与其他程序设计语言一样，学习正则表达式的关键是实践，实践，再实践。

8

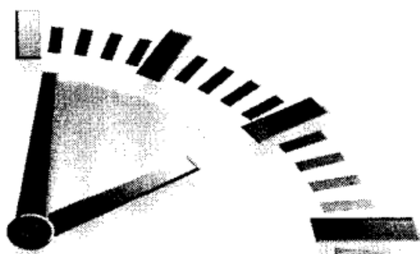


注意　我们强烈建议大家在学习本书的过程中能够亲自实践每一个示例。

1.6　小结

正则表达式是文本处理方面功能最强大的工具之一。正则表达式语言用来构造正则表达式（最终构造出来的字符串就称为正则表达式），正则表达式用来完成搜索和替换操作。

9



第2章

匹配单个字符

在本章里，你将学习如何对一个或多个字符进行简单的字符匹配。

2.1 匹配纯文本

Ben是一个正则表达式。因为本身是纯文本，所以看起来可能不像是一个正则表达式，但它的确是。正则表达式可以包含纯文本（甚至可以只包含纯文本）。当然，像这样使用正则表达式是一种浪费，但把它作为我们学习正则表达式的起点还是很不错的。

我们来看一个例子：

文本

```
Hello, my name is Ben. Please visit
my website at http://www.forta.com/.
```

正则表达式

```
Ben
```

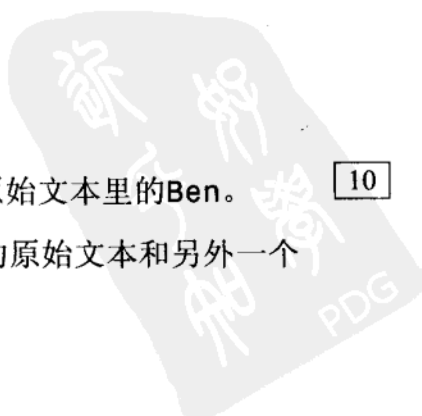
结果

```
Hello, my name is Ben. Please visit
my website at http://www.forta.com/.
```

分析

这里使用的正则表达式是纯文本，它将匹配原始文本里的Ben。

我们再来看一个例子，它使用了与刚才相同的原始文本和另外一个正则表达式：



文本

```
Hello, my name is Ben. Please visit
my website at http://www.forta.com/.
```

正则表达式

```
my
```

结果

```
Hello, my name is Ben. Please visit
my website at http://www.forta.com/.
```

分析

my也是静态文本，它在原始文本里找到了两个匹配结果。

2.1.1 有多个匹配结果

绝大多数正则表达式引擎的默认行为是只返回第1个匹配结果。具体到上面那个例子，原始文本里的第1个my通常是一个，但第2个往往不是。

怎样才能把两个或更多个匹配结果都找出来呢？绝大多数正则表达式的实现都提供了一种能够把所有的匹配结果全部找出来的机制（通常返回为一个数组或是其他的专用格式）。比如说，在JavaScript里，可选的g（意思是“global”，全局）标志将返回一个包含着所有匹配的结果数组。



注意　如果你想知道在你正在使用的语言或工具里如何进行全局匹配，请参阅本书的附录A。

11

2.1.2 字母的大小写问题

正则表达式是区分字母大小写的，所以Ben不匹配ben。不过，绝大多数正则表达的式实现也支持不区分字母大小写的匹配操作。比如说，JavaScript用户可以用i标志来强制执行一次不区分字母大小写的搜索。



注意　如果你想知道你正在使用着的语言或工具里如何进行不区分字母大小写的搜索操作，请参阅本书的附录A。

2.2 匹配任意字符

前面见到的正则表达式都是些静态的纯文本，它们根本体现不出正则表达式的威力。下面，我们一起来看看如何使用正则表达式去匹配不可预知的字符。

在正则表达式里，特殊字符（或字符集合）用来给出要搜索的东西。.字符（英文句号）可以匹配任何一个单个的字符。



提示 如果你曾经使用过DOS的文件搜索功能，你将发现正则表达式里的.字符相当于DOS的?字符。SQL用户将注意到正则表达式里的.字符相当于SQL中的_（下划线）字符。

于是，用正则表达式c.t进行的搜索将匹配到cat和cot（还能匹配到一些毫无意义的单词）。

文本

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

12

正则表达式

```
sales.
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```



分析

正则表达式sales.将把由字符串sales和另外一个字符构成的文件名查找出来。9个文件里有3个与这个模式（pattern）相匹配。



提示 人们常用模式表示实际的正则表达式。



注意 正则表达式可以用来匹配包含着字符串内容的模式。匹配的并不总是整个字符串，而是与某个模式相匹配的字符——即使它们只是整个字符串的一部分。在上面的例子里，我们使用的正则表达式并不能匹配整个文件名，它只匹配了文件名的一部分。如果你需要把某个正则表达式的匹配结果传递到其他代码或应用程序里做进一步处理，就必须记住这一细节差异。

13

.字符可以匹配任何单个的字符、字母、数字甚至是.字符本身。

文本

```
sales.xls
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

正则表达式

```
sales.
```

结果

```
sales.xls
sales1.xls
orders3.xls
sales2.xls
sales3.xls
```



```
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

分析

这个例子比上一个多了一个sales.xls文件。因为.能够匹配任何一个单个的字符，所以这个文件也与模式sales.相匹配。

在同一个正则表达式里允许使用多个.字符，它们既可以连续出现（一个接着一个——..将匹配任意两个字符），也可以间隔着出现在模式的不同位置。

我们再来看一个使用了相同原始文本的例子：把以na（表示北美）或sa（表示南美）开头的文件（不管它们后面跟着一个什么数字）找出来。 14

文本

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

正则表达式

```
.a.
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```



分析

正则表达式.a.把na1、na2和sa1找了出来，但它同时还找到了4个预料之外的匹配结果。为什么会这样？因为我们使用的模式将与第2个字符是a的任意3个字符相匹配。

我们真正需要的是后面再紧跟着一个英文句号的.a.的模式。我们再来试一次：

文本

15

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

正则表达式

```
.a..
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

分析

.a..并不比.a.好多少；新增加的.将匹配任何一个多出来的字符（不管它是什么）。既然.是一个能够与任何一个单个字符相匹配的特殊字符，我们怎样才能搜索.本身呢？

2.3 匹配特殊字符

.字符在正则表达式里有着特殊的含义。如果模式里需要一个.，就

要想办法来告诉正则表达式你需要的是.字符本身而不是它在正则表达式里的特殊含义。为此，你必须在.的前面加上一个\（反斜杠）字符来对它进行转义。\是一个元字符（metacharacter，表示“这个字符有特殊含义，而不是字符本身含义”）。

我们再来验证一次刚才的例子，这次我们使用了\对.进行转义：

16

文本

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

正则表达式

```
.a.\.xls
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
```

分析

.a.\.xls解决了问题。第1个 .匹配n（在前两个匹配结果里）或s（在第3个匹配结果里），第2个.匹配1（在第1个和第3个匹配结果里）或2（在第2个匹配结果里）。接下来，\ .匹配文件名与扩展名之间的分隔符.本身，最后的xls匹配它本身。（事实上，即使没有最后面的xls，这次搜索的结果也会与我们预想的一样；加上xls可以避免匹配到诸如sa3.doc之类的文件名。）

在正则表达式里，\字符永远出现在一个有着特殊含义的字符序列的

开头，这个序列可以由一个或多个字符构成。刚才看到的是\ .序列，在后面的章节里还会看到更多使用了\字符的例子。

我们将在第4章里对特殊字符的用法做专题讲解。



注意　如果需要搜索\本身，就必须对\字符进行转义；相应的转义序列是两个连续的反斜杠字符\\。

17



提示　我们刚才讲过，.可以匹配任何一个字符，这一说法并非绝对准确。在绝大多数的正则表达式实现里，.只能匹配除换行符以外的任何单个字符。

2.4　小结

正则表达式经常被简称为模式，它们其实是一些由字符构成的字符串。这些字符可以是普通字符（纯文本）或元字符（有特殊含义的特殊字符）。在这一章里，我们介绍了如何使用普通字符和元字符去匹配单个的字符。.可以匹配任何字符。\用来对字符进行转义。在正则表达式里，有特殊含义的字符序列总是以\字符开头。

18



第3章

匹配一组字符

在本章里，你将学习如何与字符集合打交道。与可以匹配任意单个字符的.字符（参见第2章）不同，字符集合只能匹配特定的字符和字符区间。

3.1 匹配多个字符中的某一个

第2章介绍的.字符可以匹配任意单个字符。在第2章的最后一个例子里，我们使用了.a来匹配na和sa。现在，如果在那份文件清单里增加了一个名为ca1.xls的文件，而你仍只想找出na和sa，你该怎么办？别忘了，.也能匹配c，所以文件名cal.xls也会被找出。

既然只想找出n和s，使用可以匹配任意字符的.显然不行——我们不需要匹配任意字符，我们只想匹配n和s这两个字符。在正则表达式里，我们可以使用元字符[和]来定义一个字符集合。在使用[和]定义的字符集合里，这两个元字符之间的所有字符都是该集合的组成部分，字符集合的匹配结果是能够与该集合里的任意一个成员相匹配的文本。

下面这个例子与第2章里的最后一个例子相似，但我们在这次的正则表达式里使用了一个字符集合：

文本

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
```

```
na1.xls
na2.xls
sa1.xls
ca1.xls
```

19

正则表达式

```
[ns]a.\.xls
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

分析

这里使用的正则表达式以[ns]开头；这个集合将匹配字符n或s（但不匹配字符c或其他字符）。[和]不匹配任何字符，它们只负责定义一个字符集合。接下来，正则表达式里的字符a将匹配一个a字符，.将匹配一个任意字符，\.将匹配.字符本身，xls将匹配字符串xls。从结果上看，这个模式只匹配了3个文件名，与我们的预期完全一致。



注意 虽然结果正确，但模式[ns]a.\.xls并不是最正确的答案。如果那份文件清单里还有一个名为usa1.xls的文件，它也会被匹配出来。这里涉及了位置匹配问题，而我们将在第6章里对此做专题讨论。



提示 正如看到的那样，对正则表达式进行测试是很有技巧的。验证某个模式能不能获得预期的匹配结果并不困难，但如何验证它不会匹配到你不想要的东西可就没那么简单了。

20

字符集合在不需要区分字母大小写（或者是只须匹配某个特定部分）

的搜索操作里比较常见。比如说：

文本

```
The phrase "regular expression" is often
abbreviated as RegEx or regex.
```

正则表达式

```
[Rr]eg[Ee]x
```

结果

```
The phrase "regular expression" is often
abbreviated as RegEx or regex.
```

分析

这里使用的模式包含着两个字符集合：[Rr]负责匹配字母R和r，[Ee]负责匹配字母E和e。这个模式可以匹配RegEx和regex，但不匹配REGEX。



提示 如果你打算进行一次不需要区分字母大小写的匹配，不使用这个技巧也能达到目的。这种模式最适合用在从全局看需要区分字母大小写，但在某个局部不需要区分字母大小写的搜索操作里。

3.2 利用字符集合区间

我们再来仔细看看那个从一份文件清单里找出特定文件的例子。我们刚才使用的模式[ns]a.\.xls还存在着另外一个问题。如果那份文件清单里有一个名为sam.xls的文件，结果会怎样？显然，因为.可以匹配所有的字符而不是仅限于数字，所以文件sam.xls也会出现在匹配结果里。

这个问题可以用一个如下所示的字符集合来解决：

文本

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
```

21

```
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

正则表达式

```
[ns]a[0123456789]\.xls
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

分析

在这个例子里，我们改用了另外一个模式，这个模式的匹配对象是：第1个字符必须是n或s，第2个字符必须是a，第3个字符可以是任何一个数字（因为我们使用了字符集合[0123456789]）。注意，文件名sam.xls没有出现在匹配结果里，这是因为m与我们给定的字符集合（10个数字）不相匹配。

在使用正则表达式的时候，会频繁地用到一些字符区间（0~9、A~Z，等等）。为了简化字符区间的定义，正则表达式提供了一个特殊的元字符——字符区间可以用-（连字符）来定义。

下面还是刚才那个例子，但我们这次使用了一个字符区间：

文本

```
sales1.xls
orders3.xls
sales2.xls
```

```
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

正则表达式

```
[ns]a[0-9]\.xls
```

结果

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

分析

模式[0-9]的功能与[0123456789]完全等价，所以这次的匹配结果与刚才那个例子完全一样。

字符区间并不仅限于数字，以下这些都是合法的字符区间：

- A-Z，匹配从A到Z的所有大写字母。
- a-z，匹配从a到z的所有小写字母。
- A-F，匹配从A到F的所有大写字母。
- A-z，匹配从ASCII字符A到ASCII字符z的所有字母。这个模式一般不常用，因为它还包含着[和^等在ASCII字符表里排列在Z和a之间的字符。

23

字符区间的首、尾字符可以是ASCII字符表里的任意字符。但在实际工作中，最常用的字符区间还是数字字符区间和字母字符区间。



提示 在定义一个字符区间的时候，一定要避免让这个区间的尾字符小于它的首字符（例如[3-1]）。这种区间是没有意义的，而且往往会让整个模式失效。



注意 -（连字符）是一个特殊的元字符，作为元字符它只能用在[和]之间。在字符集合以外的地方，-只是一个普通字符，只能与-本身相匹配。因此，在正则表达式里，-字符不需要被转义。

在同一个字符集合里可以给出多个字符区间。比如说，下面这个模式可以匹配任何一个字母（无论大小写）或数字，但除此以外的其他字符（既不是数字也不是字母的字符）都不匹配：

```
[A-Za-z0-9]
```

这个模式是下面这个字符集合的简写形式：

```
[ABCDEFGHIJKLMNOPQRSTUVWXYZabcde
➥fghijklmnopqrstuvwxyz01234567890]
```

正如大家看到的那样，字符范围使得正则表达式的语法变得非常简明。

下面是另一个例子，这次要查找的是RGB值（用一个十六进制数字给出的红、绿、蓝三基色的组合值，计算机可以根据RGB值把有关的文字或图象显示为由这三种颜色按给定比例调和出来的色彩）。在网页里，RGB值是以#000000（黑色）、#FFFFFF（白色）、#FF0000（红色）的形式给出的。RGB值用大写或小写字母给出均可，所以#FF00ff（品红色）也是合法的RGB值。下面就是这个例子：

文本

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

正则表达式

```
#[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]
➥[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]
```



结果

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

分析

这里使用的模式以普通字符#开头，随后是6个同样的`[0-9A-Fa-f]`字符集合。这将匹配一个由字符#开头，然后是6个数字或字母A到F（大小写均可）的字符串。

3.3 取非匹配

字符集合通常用来指定一组必须匹配其中之一的字符。但在某些场合，我们需要反过来做，给出一组不需要得到的字符。换句话说，除了那个字符集合里的字符，其他字符都可以匹配。

最先想到的办法是，用一个字符集合把你需要的字符一一列举出来，但如果只需要把一小部分字符排除在外的话，那么做既麻烦又容易有遗漏。其实这里有一个更简明的办法：用元字符^来表明你想对一个字符集合进行取非匹配——这与逻辑非运算很相似，只是这里的操作数是字符集合而已。

文本

```
sales1.xls
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

25



正则表达式

```
[ns]a[^0-9]\.xls
```

结果

```
sales1.xls
```

```
orders3.xls
sales2.xls
sales3.xls
apac1.xls
europe2.xls
sam.xls
na1.xls
na2.xls
sa1.xls
ca1.xls
```

分析

这个例子里使用的模式与前面的例子里使用的模式刚好相反。前面[0-9]只匹配数字，而这里[^0-9]匹配的是任何不是数字的字符。也就是说，[ns]a[^0-9]\.xls将匹配sam.xls，但不匹配na1.xls、na2.xls或sa1.xls。

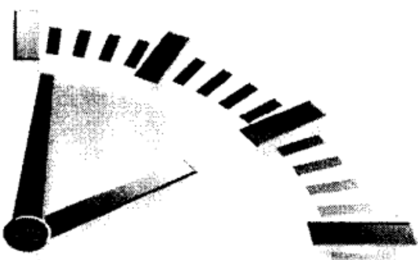


注意　^的效果将作用于给定字符集合里的所有字符或字符区间，而不是仅限于紧跟在^字符后面的那一个字符或字符区间。

3.4　小结

元字符[和]用来定义一个字符集合，其含义是必须匹配该集合里的字符之一。定义一个字符集合的具体做法有两种：一是把所有的字符都列举出来；二是利用元字符-以字符区间的方式给出。字符集合可以用元字符^来求非；这将把给定的字符集合强行排除在匹配操作以外——除了该字符集合里的字符，其他字符都可以被匹配。

26



第4章 使用元字符

本书第一次提到元字符的章节是第2章。在这一章里，你们将学习如何使用更多的元字符去匹配特定的字符或字符类型。

4.1 对特殊字符进行转义

在介绍其他元字符的用法之前，我们认为应该先把特殊字符的转义问题向大家解释清楚。

元字符是一些在正则表达式里有着特殊含义的字符。英文句号（.）是一个元字符，它可以用来匹配任何一个单个字符（详见第2章）。类似地，左方括号（[）也是一个元字符，它标志着一个字符集合的开始（详见第3章）。

因为元字符在正则表达式里有着特殊的含义，所以这些字符就无法用来代表它们本身。比如说，你不能使用一个[来匹配[本身，也不能使用.来匹配.本身。来看一个例子，我们打算用一个正则表达式去匹配一个包含着[和]字符的JavaScript数组：

文本

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

正则表达式

```
myArray[0]
```

结果

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

分析

在这个例子里，原始文本是一段JavaScript代码，正则表达式则是程序员在使用一个文本编辑器去编写JavaScript代码时经常会用到的搜索字符串。我们的本意是用这个正则表达式把代码里的myArray[0]记号找出来，可结果与预期完全不一样。为什么会这样？因为[和]在正则表达式里是用来定义一个字符集合（而不是[和]本身）的元字符，所以，myArray[0]将匹配myArray后面跟着一个该集合成员的情况，而那个集合只有一个成员0。因此，myArray[0]只能匹配到myArray0。

正如我们在第2章里解释的那样，在元字符的前面加上一个反斜杠就可以对它进行转义——转义序列\.将匹配.本身，转义序列\[将匹配[本身。每个元字符都可以通过给它加上有个反斜杠前缀的办法来转义，如此得到的转义序列将匹配那个字符本身而不是它特殊的元字符含义。要想匹配[和]，就必须对这两个字符进行转义。下面的例子与刚才的问题完全一样，但我们这次对正则表达式里的元字符都进行了转义：

文本

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

正则表达式

28

```
myArray\[0\]
```

结果

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```



分析

这次搜索取得了预期的结果。\[将匹配[，\]将匹配]，所以myArray\[0\]匹配到了myArray[0]。

具体到这个例子，用一个正则表达式来进行搜索多少有点儿小题大做——因为一个简单的文本匹配操作已足以完成这一任务，而且还会更容易一些。但如果你想查找的不仅仅是myArray[0]，还包括了myArray[1]、myArray[2]等，用一个正则表达式来进行搜索就很有必要了。具体做法是，对[和]进行转义，再列出需要在它们之间得到匹配的字符。如果你想匹配数组元素0到9，你构造出来的正则表达式应该是下面这个样子：

```
myArray\[[0-9]\]
```



提示 任何一个元字符都可以通过给它加上一个反斜杠字符(\)作为前缀的办法来转义，能够被转义的元字符并不仅局限于我们这里提到的那几个。



警告 配对的元字符(比如[或])不用作元字符时必须被转义，否则正则表达式分析器很可能会抛出一个错误。

对元字符进行转义需要用到\字符。这意味着\字符也是一个元字符——它的特殊含义是对其他元字符进行转义。正如你在第2章里看到的那样，在需要匹配\本身的时候，我们必须把它转义为\\。

29

看看下面这个简单的例子。例子中的原始文本是一个包含着反斜杠字符的文件路径（用于DOS和Windows系统），而我们想在一个Linux或Unix系统上使用这个路径——也就是说，我们需要把这个路径里的反斜杠字符（\）全部替换为正斜杠字符（/）：

文本

```
\home\ben\sales\
```

正则表达式

```
\\
```

结果

```
\home\ben\sales\
```

分析

\\匹配\，总共找到了4个匹配。如果你在这个正则表达式里只写出了一个\的话，你应该会看到一条出错消息。这是因为正则表达式分析器会认为你的正则表达式不完整，在一个完整的正则表达式里，字符\的后面永远跟着另一个字符。

4.2 匹配空白字符

元字符大致可以分为两种：一种是用来匹配文本的（比如.），另一种是正则表达式的语法所要求的（比如[和]）。随着学习的深入，你将发现越来越多的这两种元字符，而我们现在要介绍给大家的是一些用来匹配各种空白字符的元字符。

在进行正则表达式搜索的时候，我们经常会遇到需要对原始文本里的非打印空白字符进行匹配的情况。比如说，我们可能需要把所有的制表符找出来，或者我们需要把换行符找出来，这类字符很难被直接输入到一个正则表达式里，但我们可以使用表4-1列出的特殊元字符来输入它们。

30

表4-1 空白元字符

元字符	说　明
[\b]	回退（并删除）一个字符（Backspace键）
\f	换页符
\n	换行符
\r	回车符
\t	制表符（Tab键）
\v	垂直制表符

我们来看一个例子。例子中的原始文本包含着一些以逗号分隔（CSV格式）的数据记录，而我们的任务是在对这些记录做进一步处理之前，先把夹杂在这些数据里的空白行去掉。我们是这么做的：

文本

```
"101","Ben","Forta"
"102","Jim","James"

"103","Roberta","Robertson"
"104","Bob","Bobson"
```

正则表达式

```
\r\n\r\n
```

结果

```
"101","Ben","Forta"
"102","Jim","James"

"103","Roberta","Robertson"
"104","Bob","Bobson"
```

分析

`\r\n`匹配一个“回车+换行”组合，有许多操作系统（比如Windows）都把这个组合用作文本行的结束标签。使用正则表达式`\r\n\r\n`进行的搜索将匹配两个连续的行尾标签，而那正是两条记录之间的空白行。

31



提示 `\r\n`是Windows所使用的文本行结束标签。Unix和Linux系统只使用一个换行符来结束一个文本行；换句话说，在Unix/Linux系统上匹配空白行只使用`\n\n`即可，不需要加上`\r`。同时适用于Windows和Unix/Linux系统的正则表达式应该包含一个可选的`\r`和一个必须被匹配的`\n`。你可以在下一章看到一个这样的例子。

一般来说，需要匹配`\r`、`\n`和`\t`（制表符）等空白字符的情况比较多见，需要匹配其他空白字符的情况要相对少一些。



注意 你已经见过不少元字符了，但你注意到它们之间的差异了吗？.和[是元字符，但前提是你没有对它们进行转义。f和n也是元字符，但前提是你对它们进行了转义。如果你没有对f和n进行转义，它们将被解释为普通字符，只能匹配它们本身。

4.3 匹配特定的字符类别

到目前为止，你已经见过如何匹配特定的字符、如何匹配任意单个字符（用.）、如何匹配多个字符中的某一个（用[和]）以及如何进行取非匹配（用^）。字符集合（匹配多个字符中的某一个）是最常见的匹配形式，而一些常用的字符集合可以用特殊元字符来代替。这些元字符匹配的是某一类别的字符（术语称之为“字符类”）。类元字符并不是必不可少的东西（你总是可以通过逐一列举有关字符或是通过定义一个字符区间的办法来匹配某一类字符），但用它们构造出来的正则表达式简明易懂，在实践中很有用。



注意 下面列出的字符类都是最基本的，几乎所有的正则表达式实现都支持它们。

4.3.1 匹配数字（与非数字）

我们在第3章讲过，[0-9]是[0123456789]的简写形式，它可以用来匹配任何一个数字。如果你想匹配的是除数字以外的其他东西，那么把这个集合“反”过来写成[^0-9]就行了。表4-2列出了用来匹配数字和非数字的类元字符。

表4-2 数字元字符

元字符	说　明
\d	任何一个数字字符（等价于[0-9]）
\D	任何一个非数字字符（等价于[^0-9]）

为了演示这些元字符的用法，我们来看一个在前面见过的例子：

文本

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

正则表达式

```
myArray\[\d\]
```

结果

```
var myArray = new Array();
...
if (myArray[0] == 0) {
...
}
```

分析

\[匹配[，\d匹配任意单个数字字符，\]匹配]，所以myArray\[\d\]匹配出myArray[0]。myArray\[\d\]是myArray\[[0-9]\]的简写形式，而后者又是myArray\[[0123456789]\]的简写形式。这个正则表达式还可以匹配myArray[1]、myArray[2]，等等（但不匹配myArray[10]）。

33



提示 正如大家看到的那样，在与正则表达式打交道的时候，同样的问题几乎总是有好几种不同的解决办法。这些办法并无优劣之分，你尽可以选择最熟悉的那种语法。



警告 正则表达式的语法是区分字母大小写的。\d匹配数字，\D与\d的含义刚好相反。接下来将看到的其他类元字符也是如此。

4.3.2 匹配字母和数字（与非字母和数字）

字母和数字——A到Z（不分大小写）、数字0到9、再加上下划线字符

(_) ——是另一种比较常用的字符集合；这些字符常见于各种名字里，如（文件名、子目录名、变量名、数据库对象名，等等）。表4-3列出了用来匹配字母数字和非字母数字的类元字符。

表4-3 字母数字元字符

元字符	说 明
\w	任何一个字母数字字符（大小写均可）或下划线字符（等价于[a-zA-Z0-9_]）
\W	任何一个非字母数字或非下划线字符（等价于[^a-zA-Z0-9_]）

下面这个例子里的原始文本是一些来自某个数据库的记录，那些记录的内容是美国和加拿大某些城市的邮政编码[①]：

34

文本

```
11213
A1C2E3
48075
48237
M1B4F2
90046
H1H2H2
```

正则表达式

```
\w\d\w\d\w\d
```

结果

```
11213
A1C2E3
48075
48237
M1B4F2
90046
H1H2H2
```

分析

在这个模式里，交替出现的\w和\d元字符将使得匹配结果里只包含加拿大城市的邮政编码。

① 美国和加拿大城市的邮政编码规则参见附录B的B.2和B.3节。——编者注



注意 在上面这个例子里，我们使用的正则表达式解决了我们的问题。但它正确吗？请大家思考一下，为什么美国的邮政编码没有被匹配出来？是因为它们只由数字构成、还是因为什么其他原因？

我们将不给出这个问题的答案，理由很简单——例子里的模式解决了问题。这里的关键是正则表达式很少有对错之分（当然，前提是它们能解决问题），我们更关心的是它们的复杂程度——而这要由模式匹配操作的精确程度来决定；如果你需要更精确的匹配，就需要构造更复杂的正则表达式。

35

4.3.3 匹配空白字符（与非空白字符）

另一种常见的字符类别是空白字符。在本章前面的内容里，我们向大家介绍了一些用来匹配某个特定的空白字符的元字符。表4-4列出了用来匹配所有空白字符的类元字符。

表4-4 空白字符元字符

元字符	说 明
\s	任何一个空白字符（等价于[\f\n\r\t\v]）
\S	任何一个非空白字符（等价于[^\f\n\r\t\v]）



注意 用来匹配退格字符的[\b]元字符是一个特例：它不在类元字符\s的覆盖范围内，当然也就没有被排除在类元字符\S的覆盖范围外。

4.3.4 匹配十六进制或八进制数值

你或许不会遇到需要通过某个特定字符的十六进制值或八进制值来匹配它的情况，但我们希望大家明白这是可以做到的。

1. 使用十六进制值

在正则表达式里，十六进制（逢16进1）数值要用前缀\x来给出。比如说，\x0A对应于ASCII字符10（换行符），其效果等价于\n。

2. 使用八进制值

在正则表达式里，八进制（逢8进1）数值要用前缀\0来给出，数值本身可以是两位或三位数字。比如说，\011对应于ASCII字符9（制表符），其效果等价于\t。

36



注意　有不少正则表达式实现还允许使用\c前缀来指定各种控制字符。比如说，\cZ将匹配Ctrl-Z。不过，在实际工作中，必须使用这种语法的情况相当少见。

4.4　使用POSIX字符类

对元字符以及各种字符集合进行的讨论，必须要提到POSIX字符类。POSIX字符类是许多（但不是所有）正则表达式实现都支持的一种简写形式。



注意　JavaScript不支持在正则表达式里使用POSIX字符类。

表4-5　POSIX字符类

字符类	说　明
[:alnum:]	任何一个字母或数字（等价于[a-zA-Z0-9]）
[:alpha:]	任何一个字母（等价于[a-zA-Z]）
[:blank:]	空格或制表符（等价于[\t][①]）
[:cntrl:]	ASCII控制字符（ASCII 0到31，再加上ASCII 127）
[:digit:]	任何一个数字（等价于[0-9]）
[:graph:]	和[:print:]一样，但不包括空格
[:lower:]	任何一个小写字母（等价于[a-z]）
[:print:]	任何一个可打印字符
[:punct:]	既不属于[:alnum:]也不属于[:cntrl:]的任何一个字符
[:space:]	任何一个空白字符，包括空格（等价于[^\f\n\r\t\v][②]）
[:upper:]	任何一个大写字母（等价于[A-Z]）
[:xdigit:]	任何一个十六进制数字（等价于[a-fA-F0-9]）

37

① 注意，字母t后有一个空格。——译者注

② 注意，字母v后有一个空格。——译者注

POSIX语法与我们此前见过的元字符不太一样。为了演示POSIX字符类的用法，我们来看一个前一章里的例子——利用正则表达式从一段HTML代码里把RGB值查找出来：

文本

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

正则表达式

```
#[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[
➥:xdigit:]]
```

结果

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

分析

在前一章里使用的模式是重复写出的6个`[0-9A-Fz-f]`字符集合，把那6个`[0-9A-Fz-f]`全部替换为`[[:xdigit:]]`就得到这里的模式。它们的匹配结果完全一样。 38



注意 这里使用的模式以`[[`开头、以`]]`结束（两对方括号）。这是使用POSIX字符类所必须的。POSIX字符类必须括在`[:`和`:]`之间，我们使用的POSIX字符类是`[:xdigit:]`（不是`:xdigit:`）。外层的`[`和`]`字符用来定义一个字符集合，内层的`[`和`]`字符是POSIX字符类本身的组成部分。



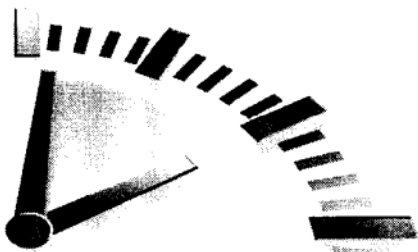
警告 一般来说，支持POSIX标准的正则表达式实现都支持表4-5所列出的那12个POSIX字符类，但在一些细节方面可能会与这里的描述有细微的差异。

4.5 小结

我们在第2章和第3章对字符匹配操作和字符集合匹配操进行了讨论。在此基础上，这一章对用来匹配特定字符（制表符、换行符，等等）和用来匹配一个字符集合或字符类（数字、字母数字字符，等等）的元字符进行了讲解。这些简短的元字符和POSIX字符类可以用来简化正则表达式模式。

39





第5章

重复匹配

在前几章里，你们学习了如何使用各种元字符、字符集合和字符类去匹配单个字符。在这一章里，你将学习如何匹配多个连续重复出现的字符或字符集合。

5.1 有多少个匹配

通过前面的学习，我们已经把正则表达式模式匹配操作的基础知识全都介绍给了大家，但我们给出的每个例子都有一个非常严格的限制。现在，请大家思考一下，如何构造一个匹配电子邮件地址的正则表达式。电子邮件地址的基本格式应该是如下所示的样子：

```
text@text.text
```

利用前一章讨论的元字符，你可能会写出一个如下所示的正则表达式：

```
\w@\w\.\w
```

\w可以匹配所有的字母和数字字符（以及下划线字符_，这个字符在电子邮件地址里是合法的）；@字符不需要被转义，但.字符需要。

这个正则表达式本身没有任何错误，可它几乎没有任何实际的用处——它只能匹配a@b.c形式的电子邮件地址（虽然在语法方面没有任何问题，但这显然不是一个合法的地址）。导致这一结果的关键是\w只能匹配单个字符，而我们无法预知电子邮件地址的各个字段会有多少个字符。举个最简单的例子，下面这些都是合法的电子邮件地址，但它们在@前面的字符个数都不一样。

```
b@forta.com
ben@forta.com
bforta@forta.com
```

40

要想解决这类问题，我们需要一种能够匹配多个字符的办法，这可以通过使用几种特殊的元字符来做到。

5.1.1　匹配一个或多个字符

要想匹配同一个字符（或字符集合）的多次重复，只要简单地给这个字符（或字符集合）加上一个+字符作为后缀就行了。+匹配一个或多个字符（至少一个；不匹配零个字符的情况）。比如，a匹配a本身，a+将匹配一个或多个连续出现的a。类似地，[0-9]匹配任意单个数字，[0-9]+将匹配一个或多个连续的数字。



提示　在给一个字符集合加上+后缀的时候，必须把+放在这个字符集合的外面。比如说，[0-9]+是正确的，[0-9+]则不是。

[0-9+]其实也是一个合法的正则表达式，但它匹配的不是一个或多个数字；它定义了一个由数字0到9和+构成的字符集合，因而只能匹配一个单个的数字字符或加号。虽然合法，可它并不是我们需要的东西。

重新回到电子邮件地址的例子，我们这次将使用+来匹配一个或多个字符：

文本

```
Send personal email to ben@forta.com. For questions
about a book use support@forta.com. Feel free to send
unsolicited email to spam@forta.com (wouldn't it be
nice if it were that simple, huh?).
```

正则表达式

```
\w+@\w+\.\w+
```

结果

```
Send personal email to ben@forta.com. For questions
about a book use support@forta.com. Feel free to send
```

41

```
unsolicited email to spam@forta.com (wouldn't it be
nice if it were that simple, huh?).
```

分析

这个模式把原始文本里的3个电子邮件地址全都正确地匹配出来了。这个正则表达式先用第一个\w+匹配一个或多个字母数字字符，再用第二个\w+匹配@后面的一个或多个字符，然后匹配一个.字符（使用转义序列\ .），最后用第三个\w+匹配电子邮件地址的剩余部分。



提示 +是一个元字符。如果需要匹配+本身，就必须使用它的转义序列\+。

+还可以用来匹配一个或多个字符集合。为了演示这种用法，我们在下面这个例子里使用了和刚才一样的正则表达式，但原始文本和上一个例子稍有不同：

文本

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

正则表达式

```
\w+@\w+\.\w+
```

结果

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

42

分析

这个正则表达式匹配到了5个电子邮件地址，但其中有两个不够完

整。为什么会这样？因为我们在构造这个正则表达式的时候只想到在@字符的后面会有一个.字符分开两个字符串的情况，没有想到在@字符的前面还会有.字符。因此，虽然ben.forta@forta.com是一个完全合法的电子邮件地址，但这个正则表达式只能匹配forta（而不是ben.forta）——别忘了，\w只能匹配字母和数字字符，不能匹配出现在字符串中间的.字符。

要想干净彻底地解决这个问题，我们需要匹配\w或.。用正则表达式的术语来说，我们需要匹配字符集合[\w\.]。下面是上面那个例子的改进版本：

文本

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

正则表达式

```
[\w.]+@[\w.]+\.\w+
```

结果

```
Send personal email to ben@forta.com or
ben.forta@forta.com. For questions about a
book use support@forta.com. If your message
is urgent try ben@urgent.forta.com. Feel
free to send unsolicited email to
spam@forta.com (wouldn't it be nice if
it were that simple, huh?).
```

分析

问题似乎得到了圆满解决。[\w.]+将匹配字符集合[\w.]（字母数字字符、下划线和.）的一次或多次重复出现，ben.forta完全符合这一条件。考虑到有些电子邮件地址会有多层域名（或主机名），我们在@字符的后面也使用了一个[\w.]+。

43



注意 这个正则表达式的最后一部分是\w+而不是[\w.]+，你知道这是为什么吗？把[\w.]用作这个模式的最后一部分会在第2、第3和第4个匹配上出问题——你不妨试试看。



注意 细心的读者可能已经注意到了：我们没有对字符集合[\w.]里的.字符进行转义。尽管如此，它还是把原始文本里的.字符匹配出来了。一般来说，当在字符集合里使用的时候，像.和+这样的元字符将被解释为普通字符，不需要被转义——但转义了也没有坏处。[\w.]的使用效果与[\w\.]是一样的。

5.1.2 匹配零个或多个字符

+匹配一个或多个字符，但不匹配零个字符——+最少也要匹配一个字符。那么，如果你想匹配一个可有可无的字符——也就是该字符可以出现零次或多次的情况，你该怎么办呢？

这种匹配需要用*元字符来完成。*的用法与+完全一样——只要把它放在一个字符（或一个字符集合）的后面，就可以匹配该字符（或字符集合）连续出现零次或多次的情况。比如说，模式B.* Forta将匹配B Forta、B. Forta、Ben Forta和其他有类似规律的组合。

为了演示+和*的区别，我们来看两个匹配电子邮件地址的例子。先看第一个：

文本

```
Hello .ben@forta.com is my email address.
```

正则表达式

```
[\w.]+@[\w.]+\.\w+
```

结果

```
Hello .ben@forta.com is my email address.
```

分析

[\w.]+将匹配字符集合[\w.]（字母数字字符、下划线和.）的一次

或多次重复出现，而.ben。完全符合这一条件。这显然是一个打字错误（原始文本里多了一个.），但这并不是我们这里最关心的问题。问题的关键在于：虽然 . 是电子邮件地址里的合法字符，但把它用作电子邮件地址的第一个字符就不合法了。

一个电子邮件地址可以有任意多个字符，但它的第一个字符必须是一个字母或数字字符。根据这一要求，我们真正需要的是一个如下例所示的模式：

文本

```
Hello .ben@forta.com is my email address.
```

正则表达式

```
\w+[\w.]*@[\w.]+\.\w+
```

结果

```
Hello .ben@forta.com is my email address.
```

分析

这个模式看起来相当复杂，但并不难理解。开头的\w+负责匹配电子邮件地址里的第一个字符（一个字母数字字符，不包括.字符）。接下来的[\w.]*负责匹配电子邮件地址里第一个字符之后、@字符之前的所有字符——这个部分可以包含零个或多个字母数字字符和.字符。至于这个模式的其他部分，我们已经在第4章里解释过了。在这个例子里，解决问题的关键是能不能想到用[\w.]*来匹配字符集合[\w.]（字母数字字符、下划线和.）的零次或多次重复出现。



注意　可以把*理解为一个用来表明这样一种含义的元字符："在我前面的字符（或字符集合）是可选的"。*与+的区别是：+匹配一个或多个字符（或字符集合），最少要匹配一次；*匹配零个或任意多个字符（或字符集合），可以没有匹配。



提示　*是一个元字符。如果需要匹配*本身，就必须使用它的转义序列*。

5.1.3 匹配零个或一个字符

另一个非常有用的元字符是?。?只能匹配一个字符（或字符集合）的零次或一次出现，最多不超过一次——请仔细体会?与+和*的相似和区别之处。如果需要在一段文本里匹配某个特定的字符（或字符集合）而该字符可能出现、也可能不出现，?无疑是最佳的选择。

请看下面这个例子：

文本

```
The URL is http://www.forta.com/, to connect
securely use https://www.forta.com/ instead.
```

正则表达式

```
http://[\w./]+
```

结果

```
The URL is http://www.forta.com/, to connect
securely use https://www.forta.com/ instead.
```

分析

这是一个用来匹配URL地址的模式：http://是普通文本，只能匹配它本身；随后的[\w./]+匹配字符集合[\w./]（字母数字字符、.和/）的一次或多次重复出现。这个模式只匹配到了第一个URL地址（以http://开头的那个），没能匹配到第二个（以https://开头的那个）。简单地在http的后面加上一个s*（s的零次或多次重复）并不能真正解决这个问题，因为那会使得httpsssss://（如此开头的URL地址显然是不合法的）也被认为是一个合法的匹配。

怎么办？看看下面这个例子就知道了——在http的后面加上一个s?：

46

文本

```
The URL is http://www.forta.com/, to connect
securely use https://www.forta.com/ instead.
```

正则表达式

```
https?://[\w./]+
```

结果

```
The URL is http://www.forta.com/, to connect
securely use https://www.forta.com/ instead.
```

分析

这个模式的开头部分是https?。?在这里的含义是：我前面的字符（s）要么不出现，要么最多出现一次。换句话说，https?://既可以匹配http://，也可以匹配https://，但也就仅此而已。

在4.3.3节里有一个用模式\r\n\r\n去匹配空白行的例子。我在分析完那个例子后说过这样的话：在Unix或Linux系统上匹配空白行只使用\n\n即可，不需要加上\r；同时适用于Windows和Unix/Linux系统的正则表达式应该包含一个可选的\r和一个必须被匹配的\n。现在，你应该想到可以用?来解决这个问题了吧？下面还是那个例子，但我们这次将使用一个略有不同的正则表达式：

文本

```
"101","Ben","Forta"
"102","Jim","James"

"103","Roberta","Robertson"
"104","Bob","Bobson"
```

正则表达式

```
[\r]?\n[\r]?\n
```

结果

```
"101","Ben","Forta"
"102","Jim","James"


"103","Roberta","Robertson"
"104","Bob","Bobson"
```

47

分析

[\r]?\n匹配一个可选的\r和一个必不可少的\n。





提示 细心的读者可能已经注意到了，上面这个例子里的正则表达式使用的是[\r]?而不是\r?。[\r]定义了一个字符集合，该集合只有元字符\r这一个成员，因而[\r]?在功能上与\r?完全等价。[ ]的常规用法是把多个字符定义为一个集合，但有不少程序员喜欢把一个字符也定义为一个集合。这么做的好处是可以增加可读性和避免产生误解，让人们一眼就可以看出哪个字符与哪个元字符相关联。这里必须提醒大家注意这样一个细节：如果你打算同时使用[ ]和?，千万记得应该把?放在字符集合的外面。具体到刚才那个匹配URL地址的例子，写成http[s]?://是正确的，若是写成http[s?]://可就弄巧成拙了。



提示 ?是一个元字符。如果需要匹配?本身，就必须使用它的转义序列\?。

5.2 匹配的重复次数

正则表达式里的+、*和?解决了许多问题，但有些问题光靠它们还不够。请思考以下问题：

- +和*匹配的字符个数没有上限。我们无法为它们将匹配的字符个数设定一个最大值。
- +、*和?至少匹配零个或一个字符。我们无法为它们将匹配的字符个数另行设定一个最小值。
- 如果只使用+和*，我们无法把它们将匹配的字符个数设定为一个精确的数字。

48

为了解决这些问题并让程序员对重复性匹配有更多的控制，正则表达式语言提供了一个用来设定重复次数（interval）的语法。重复次数要用{和}字符来给出——把数值写在它们之间。



注意　{和}是元字符。如果需要匹配{和}本身，就应该用\对它们进行转义。不过，即使你没有对{和}进行转义，大部分正则表达式实现也能正确地处理它们（根据具体情况把它们解释为普通字符或元字符）。话虽如此，为了避免不必要的麻烦，你最好不要依赖这种行为；在需要把{和}当做普通字符来匹配的场合，还是使用它们的转义序列\{和\}比较稳妥。

5.2.1　为重复匹配次数设定一个精确的值

如果你想为重复匹配次数设定一个精确的值，把那个数字写在\{和\}之间即可。比如说，{3}意味着模式里的前一个字符（或字符集合）必须在原始文本里连续重复出现3次才算是一个匹配；如果只重复了两次，则不算是一个匹配。

为了演示这种用法，我们再来看一下匹配RGB值的例子（请对照第3章和第4章里的类似例子）。你应该记得，RGB值是一个十六进制数值，这个值分成3个部分，每个部分包括两位十六数字。下面是我们在第3章里用来匹配RGB值的模式：

```
#[0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]
➥[0-9A-Fa-f]
```

下面是我们在第4章里用来匹配RGB值的模式，它使用了POSIX字符类：

```
#[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[
➥:xdigit:]]
```

这两个模式本身并无不妥，但美中不足的是你不得不重复写出6次相同的字符集合（或POSIX字符类）。下面是一个同样的例子，但我们这次将使用{ }语法来明确指定一个重复次数：

49

文本

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

正则表达式

```
#[[:xdigit:]]{6}
```



结果

```
<BODY BGCOLOR="#336633" TEXT="#FFFFFF"
      MARGINWIDTH="0" MARGINHEIGHT="0"
      TOPMARGIN="0" LEFTMARGIN="0">
```

分析

[:xdigit]匹配一个十六进制数字，{6}要求这个POSIX字符类必须连续出现6次。类似地，使用模式#[0-9A-Fa-f]{6}也可以解决这个问题。

5.2.2 为重复匹配次数设定一个区间

{}语法还可以用来为重复匹配次数设定一个区间——也就是为重复匹配次数设定一个最小值和一个最大值。这种区间必须以{2, 4}这样的形式给出——{2, 4}的含义是最少重复2次、最多重复4次。在下面的例子里，我们将使用一个这样的正则表达式来检查日期的格式：

文本

```
4/8/03
10-6-2004
2/2/2
01-01-01
```

正则表达式

```
\d{1,2}[-\/]\d{1,2}[-\/]\d{2,4}
```

结果

```
4/8/03
10-6-2004
2/2/2
01-01-01
```

50

分析

这里列出的日期是一些由用户通过某个表单字段输入的值——在对这些日期值做进一步处理之前，我们需要先检查它们的格式是否正确。\d{1, 2}将匹配一个或两个数字字符（用来匹配日子和月份）；\d{2, 4}用来匹配年份；[-\/]（请注意，这个\/其实是一个\和一个/）用来匹配日期值里的分隔符-或/。我们总共匹配到了3个日期值，但2/2/2不在此列（因为它的年份太短了）。



提示　在这个例子里，我们使用了/的转义序列\/。这在许多正则表达式实现里是不必要的，但有些正则表达式分析器要求我们必须这样做。为避免不必要的麻烦，在需要匹配/字符本身的时候，你最好总是使用它的转义序列。

注意，上面这个例子里的模式并不能检查日期值是否有效；诸如54/67/9999之类的无效日期也能通过这一测试。它只能用来检查日期值的格式是否正确（这一环节通常安排在日期值本身的有效性检查之前）。



注意　重复次数可以是0。比如，{0, 3}表示重复次数可以是0、1、2或3。

我们曾经讲过，?匹配它之前一个字符（或字符集合）的零次或一次出现。因此，从效果上看，?等价于{0, 1}。

5.2.3　匹配“至少重复多少次”

{ }语法的最后一种用法是给出一个最小的重复次数（但不必给出一个最大值）。{ }的这种用法与我们用来为重复匹配次数设定一个区间的{ }语法很相似，只是省略了最大值部分而已。比如说，{3, }表示至少重复3次，与之等价的说法是“必须重复3次或更多次”。

51

我们来看一个综合了本章主要内容的例子。在这个例子里，我们使用一个正则表达式把所有大于或等于$100美元的金额找出来：

文本

```
1001: $496.80
1002: $1290.69
1003: $26.43
1004: $613.42
1005: $7.61
1006: $414.90
1007: $25.00
```



正则表达式

```
\d+: \$\d{3,}\.\d{2}
```

结果

```
1001: $496.80
1002: $1290.69
1003: $26.43
1004: $613.42
1005: $7.61
1006: $414.90
1007: $25.00
```

分析

这个例子里的原始文本来自一份报表，它的第一列是定单号，第二列是定单金额。我们构造的正则表达式首先使用了一个`\d+:`来匹配定单号（这部分其实可以省略——我们可以只匹配金额部分而不是匹配包括定单号在内的一整行）。模式`\$\d{3,}\.\d{2}`用来匹配金额部分：`\$`匹配`$`、`\d{3,}`匹配至少3位数字（也就是所有大于或等于$100美元的金额）、`\.`匹配`.`、`\d{2}`匹配小数点后面的两位数字。整个模式从7条记录里正确地匹配到了4条符合要求的记录。



提示 在进行这种重复次数匹配的时候一定要小心。如果你遗漏了花括号里的逗号`,`，你的模式将变成（具体到这个例子）精确匹配3位数字而不再是匹配至少3位数字。

52

5.3 防止过度匹配

`?`只能匹配零个或一个字符，`{n}`和`{m,n}`也有一个重复次数的上限；换句话说，这几种语法所定义的“重复次数”都是有限的。但本章介绍的其他重复匹配语法在重复次数方面都没有上限值，而这样做有时会导致过度匹配的现象。

到目前为止，我们选用的例子都不存在过度匹配的问题，但你迟早会遇到类似于下面这个例子的情况。这个例子里的原始文本来自一个Web页面，其中包含着两个HTML <B>标签；而我们的任务是用一个正则表达

式把那两个<B>标签里的文本匹配出来（为了对这些文本进行替换或排版等）。下面就是这个例子：

文本

```
This offer is not available to customers
living in <B>AK</B> and <B>HI</B>.
```

正则表达式

```
<[Bb]>.*</[Bb]>
```

结果

```
This offer is not available to customers
living in <B>AK</B> and <B>HI</B>.
```

分析

<[Bb]>匹配<B>标签（大小写均可），</[Bb]>匹配</B>标签（也是大小写均可）。但这个模式只找到了一个匹配而不是预期中的两个：第一个<B>标签之后、最后一个</B>标签之前的所有东西——AK</B> and <B>HI——被.*一网打尽。虽然没有漏掉我们想要匹配的文本，但问题是第2个<B>标签不明不白地“失踪”了。

为什么会这样？因为*和+都是所谓的“贪婪型”元字符，它们在进行匹配时的行为模式是多多益善而不是适可而止的。它们会尽可能地从一段文本的开头一直匹配到这段文本的末尾，而不是从这段文本的开头匹配到碰到第一个匹配时为止。

53

在不需要这种“贪婪行为”的时候该怎么办？答案是使用这些元字符的“懒惰型”版本（“懒惰”在这里的含义是匹配尽可能少的字符——与“贪婪型”元字符的行为模式刚好相反）。懒惰型元字符的写法很简单，只要给贪婪型元字符加上一个?后缀即可。表5-1列出了几个常用的贪婪型元字符和它们的懒惰型版本。

表5-1 常用的贪婪型元字符和它们的懒惰型版本

贪婪型元字符	懒惰型元字符
*	*?
+	+?
{n, }	{n, }?

*?是*的懒惰型版本；下面是使用*?来解决刚才那个例子的做法：

文本

```
This offer is not available to customers
living in <B>AK</B> and <B>HI</B>.
```

正则表达式

```
<[Bb]>.*?</[Bb]>
```

结果

```
This offer is not available to customers
living in <B>AK</B> and <B>HI</B>.
```

分析

问题得到了圆满解决。因为使用了懒惰的*?，第一个匹配将仅限于AK，原始文本里的<B>HI</B>成为了第二个匹配。

54



注意 这本书里的大多数例子使用的都是“贪婪型”元字符，而我们这么做的出发点是为了让那些示例模式尽可能地简明易懂。在实际工作中，请务必根据具体情况来选用“贪婪型”或“懒惰型”元字符。

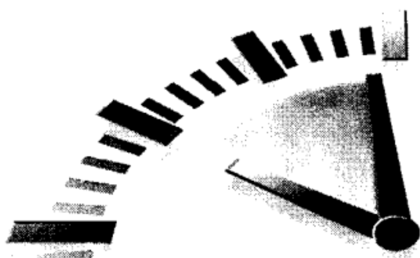
5.4 小结

正则表达式的真正威力体现在重复次数匹配方面。本章介绍了+（匹配字符或字符集合的一次或多次重复出现）、*（匹配字符或字符集合的零次或多次重复出现）、?（匹配字符或字符集合的零次或一次出现）等几个元字符的用法。要想获得更精确的控制，你可以用{ }语法来精确地设定一个重复次数或是重复次数的最小值和最大值。元字符分“贪婪型”和“懒惰型”两种；在需要防止过度匹配的场合下，请使用“懒惰型”元字符来构造你的正则表达式。

55

第6章

位置匹配



到目前为止，你已经学习了许多元字符的用法。只要灵活运用这些知识，你就可以对任意字符（或字符集合）及其各种组合和重复进行匹配——那些字符（或字符集合）可以出现在原始文本里的任意位置。可是，在某些场合，你需要且只需要对某段文本的特定位置进行匹配，这就引出了位置匹配的概念，而这个概念正是本章的学习重点。

6.1 边界

位置匹配用来解决在什么地方进行字符串匹配操作的问题。为了让大家对位置匹配及其相关概念有一个直观的认识，我们先来看一个例子：

文本

```
The cat scattered his food all over the room.
```

正则表达式

```
cat
```

结果

```
The cat scattered his food all over the room.
```

分析

模式cat把原始文本里的所有cat都找了出来，单词scattered里的那个cat也不例外。但这一结果并不是我们所预期的，我们只想把单词cat本身找出来。我们本想用这种办法把所有的cat替换为dog，但得到的结果却是一个毫无实际意义的句子：

```
The dog sdogtered his food all over the room.
```

能够正确解决这个问题的办法只有一个：使用边界限定符，也就是在正则表达式里用一些特殊的元字符来表明我们想让匹配操作在什么位置（或边界）发生。

56

6.2 单词边界

第一种边界（也是最常用的边界）是由限定符\b指定的单词边界。顾名思义[①]，\b用来匹配一个单词的开始或结尾。

为了演示\b的用法，让我们回到刚才的例子再做一次尝试，但我们这次将用上单词边界：

文本

```
The cat scattered his food all over the room.
```

正则表达式

```
\bcat\b
```

结果

```
The cat scattered his food all over the room.
```

分析

在原始文本里，单词cat的前后都有一个空格，而这将与模式\bcat\b相匹配（空格是用来分隔单词的字符之一）。单词scattered中的字符序列cat不能与这个模式相匹配，因为它的前一个字符是s、后一个字符是t（这两个字符都不能与\b相匹配）。



注意 \b到底匹配什么东西呢？正则表达式引擎不懂英语（事实上，它不懂任何人类语言），也不知道什么是单词边界。简单地说，\b匹配的是一个这样的位置，这个位置位于一个能够用来构成单词的字符（字母、数字和下划线，也就是与\w相匹配的字符）和一个不能用来构成单词的字符（也就是与\W相匹配的字符）之间。

① b是英文boundary（边界）的首字母。——编者注

这里要特别注意的是，如果你想匹配一个完整的单词，就必须在你57想要匹配的文本的前后都加上\b限定符。请看下面这个例子：

文本

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

正则表达式

```
\bcap
```

结果

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

分析

模式\bcap将匹配以字符序列cap开头的任何一个单词。这里总共找到了4个匹配，其中有3个是以字符序列cap开头的其他单词而不是单词cap本身。

下面这个例子里的原始文本还是刚才那段文字，但在这次的正则表达式里只有一个后缀的\b限定符：

文本

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

正则表达式

```
cap\b
```

结果

```
The captain wore his cap and cape proudly as
he sat listening to the recap of how his
crew saved the men from a capsized vessel.
```

分析

模式cap\b将匹配以字符序列cap结束的任何一个单词。这里总共找到了582个匹配，其中一个是以字符序列cap结束的其他单词而不是单词cap本身。

如果你只想匹配单词cap本身，就必须使用\bcap\b做为模式，它才是你需要的正确答案。



注意 \b匹配且只匹配一个位置，不匹配任何字符。用\bcat\b匹配到的字符串的长度是3个字符（c、a、t），不是5个字符。

如果你想表明不匹配一个单词边界[①]，请使用\B。在下面的例子里，我们将使用\B来查找其前后都有多余空格的连字符：

文本

```
Please enter the nine-digit id as it
appears on your color - coded pass-key.
```

正则表达式

```
\B-\B
```

结果

```
Please enter the nine-digit id as it
appears on your color - coded pass-key.
```

分析

\B-\B将匹配一个前后都不是单词边界的连字符。nine-digit和pass-key中的连字符不能与之匹配，但color-coded中的连字符可以与之匹配[②]。

正如我们在第4章里见到的那样，同一个元字符的大写形式与它的小写形式在功能上往往刚好相反。

59



注意 除了用来匹配单词边界（开头或结束均可）的\b，有些正则表达式实现还支持另外两个元字符：\<只匹配单词的开头；\>只匹配单词的结束。不过，虽然这两种元字符可以提供粒度更细的控制，但支持它们的正则表达式引擎却并不多见（据笔者所知，egrep程序是支持\<和\>的，但许多其他文本匹配工具则不支持它们）。

---

① 即字母数字下划线之间，或者非字母数字下划线之间。——编者注

② 因为空格和连字符都不是字母数字或下划线。——编者注

6.3 字符串边界

单词边界可以用来进行与单词有关的位置匹配（单词的开头、单词的结束、整个单词，等等）。字符串边界有着类似的用途，只不过是用来进行与字符串有关的位置匹配而已（字符串的开头、字符串的结束、整个字符串，等等）。用来定义字符串边界的元字符有两个：一个是用来定义字符串开头的^，另一个是用来定义字符串结尾的$。



注意　还记得吗？我们在第3章里已经见识过元字符^了，但那时的它是一个用来对字符集合进行“求非”操作的元字符。那它还怎么用来表明一个字符串的开头呢？

^是几个有着多种用途的元字符之一。只有当它出现在一个字符集合里（被放在[和]之间）并紧跟在左方括号[的后面时，它才能发挥“求非”作用。如果是在一个字符集合的外面并位于一个模式的开头，^将匹配字符串的开头。

为了演示字符串边界的用法，我们在下面准备了一个例子。合法的XML文档都必须以<?xml>标签开头并有一些其他属性（比如一个版本号，如<?xml version="1.0" ?>）。下面这个简单的测试可以检查一段文本是否是一篇XML文档：

60

文本

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

正则表达式

```
<\?xml.*\?>
```

结果

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

分析

这个模式似乎能够解决问题：`<\?xml`匹配`<?xml`，`.*`匹配随后的任意文本（`.`的零次或多次重复出现），`\?>`匹配`?>`。

这是一个非常不准确的测试。在下面的例子里，上例中的模式虽然匹配到了一个XML文档的开头部分，但位置却完全不对。它匹配到的语句位于文档的第2行而不是第1行。

文本

```
This is bad, real bad!
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

正则表达式

```
<\?xml.*\?>
```

结果

```
This is bad, real bad!
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

分析

模式`<\?xml\?>`匹配到的是整个文本的第2行。虽然它也是XML文档的开始标签，但因为出现在文本的第2行，所以这份文档肯定不是一份合法的XML文档，把它当做一份XML文档来处理会导致种种问题。

61

这里需要的是一个能够确保被匹配到的`<?xml>`标签出现在字符串最开始处的测试，而这正是`^`元字符大显身手的地方；如下所示：

文本

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

正则表达式

```
^\s*<\?xml.*\?>
```

结果

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://tips.cf"
xmlns:impl="http://tips.cf" xmlns:intf="http://tips.cf"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

分析

^匹配一个字符串的开头位置，所以^\s*将匹配一个字符串的开头位置和随后的零个或多个空白字符（这解决了<?xml>标签前允许有空格、制表符、换行符等空白字符的问题）。作为一个整体，模式^\s*<\?xml.*\?>不仅能正确地匹配一个位置正确的<?xml>标签，还能对合法的空白字符做出妥善处理。



提示　虽然模式^\s*<\?xml.*\?>解决了上例中的问题，但那只是因为这个例子里的原始文本并不完整而已。如果这段原始文本是一份完整的XML文档，这个例子将变成一个“贪婪型”元字符的典型示例。还好，我们已经知道解决“贪婪型”元字符问题的最佳办法是把.*替换为.*?。

62

除了位置上的差异，$的用法与^完全一样。比如说，在一份Web页面里，</html>标签的后面不应该再有任何实际内容，而这一点可以用下面这个模式来检查：

正则表达式

```
</[Hh][Tt][Mm][Ll]>\s*$
```

分析

我们用了4个字符集合来分别匹配H、T、M、L等4个字符（这样就可以对这几个字符的各种大小写组合形式进行处理了），\s*$匹配一个字符串结尾处的零个或多个空白字符。



注意　模式^.*$是一个在语法上完全正确的正则表达式；它几乎总能找到一个匹配，但没有任何实际用途。你能分析出这个模式将匹配什么以及它在什么情况下会找不到任何匹配吗？

分行匹配模式

我们刚刚讲过，^匹配一个字符串的开头，$匹配一个字符串的结尾。但这一结论并非绝对正确，它还有一个例外或者说有一种改变这种行为的办法。

有许多正则表达式都支持使用一些特殊的元字符去改变另外一些元字符行为的做法，用来启用分行匹配模式（multiline mode）的(?m)记号就是一个能够改变其他元字符行为的元字符序列。分行匹配模式将使得正则表达式引擎把行分隔符当做一个字符串分隔符来对待。在分行匹配模式下，^不仅匹配正常的字符串开头，还将匹配行分隔符（换行符）后面的开始位置（这个位置是不可见的）；类似地，$不仅匹配正常的字符串结尾，还将匹配行分隔符（换行符）后面的结束位置。

在使用时，(?m)必须出现在整个模式的最前面，就像下面这个例子里那样。在这个例子里，我们将使用一个正则表达式把一段JavaScript代码里的注释内容全部查找出来：

文本

63

```
<SCRIPT>
function doSpellCheck(form, field) {
   // Make sure not empty
   if (field.value == '') {
      return false;
   }
   // Init
   var windowName='spellWindow';
   var
spellCheckURL='spell.cfm?formname=comment&fieldname='+field.
➥name;
...
   // Done
   return false;
}
</SCRIPT>
```

正则表达式

```
(?m)^\s*//.*$
```



结果

```
<SCRIPT>
function doSpellCheck(form, field) {
   // Make sure not empty
   if (field.value == '') {
      return false;
   }
   // Init
   var windowName='spellWindow';
   var
spellCheckURL='spell.cfm?formname=comment&fieldname='+field.
➥name;
...
   // Done
   return false;
}
</SCRIPT>
```

分析

`^\s*//.*$`将匹配一个字符串的开始，然后是任意多个空白字符，再后面是`//`（JavaScript代码里的注释标签），再往后是任意文本，最后是一个字符串的结束。不过，这个模式只能找出第一条注释（并认为这条注释将一直延续到文件的末尾，因为`*`是一个“贪婪型”元字符）。加上`(?m)`前缀之后，`(?m)^\s*//.*$`将把换行符视为一个字符串分隔符，这样就可以把每一行注释都匹配出来了。

64



警告 有许多正则表达式实现不支持`(?m)`。



注意 有些正则表达式实现还支持使用`\A`来定义一个字符串的开始，以`\Z`来定义一个字符串的结束的做法。此时，`\A`和`\B`的作用将基本等价于`^`和`$`，但请注意，`\A`和`\B`不会因为加上了`(?m)`前缀而改变行为。换句话说，在跨行匹配模式下使用`\A`和`\B`的做法不会收到在分行匹配模式下使用`^`和`$`的效果。

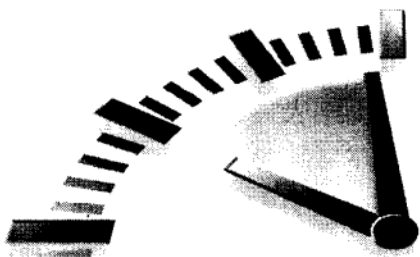
6.4 小结

正则表达式不仅可以用来匹配任意长度的文本块，还可以用来匹配出现在字符串中特定位置的文本。\b用来指定一个单词边界（\B刚好相反）。^和$用来指定字符串边界（字符串的开头和字符串的结束）。如果与(?m)配合使用，^和$还将匹配在一个换行符处开头或结束的字符串（此时，换行符将被视为一个字符串分隔符）。

65



第 7 章



使用子表达式

元字符和字符是正则表达式的基本构件，它们的用法我们已经在此前的章节里演示过了。在这一章里，你们将学习如何运用子表达式（subexpression）的概念对表达式进行分组和归类。

7.1 什么是子表达式

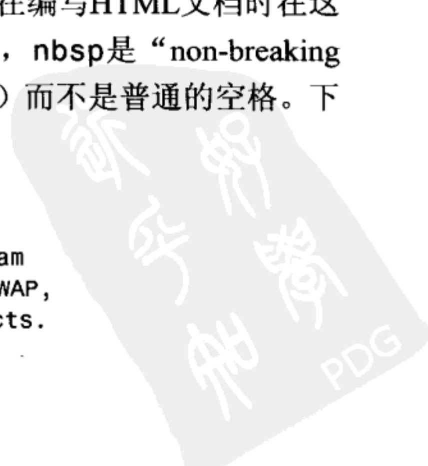
我们在第5章学习了如何匹配一个字符的连续多次重复。正如我们讨论的那样，`\d+`将匹配一个或多个数字字符，而`https?://`将匹配`http://`或`https://`。

在这两个例子里（事实上，是在我们此前见过的所有例子里），用来表明重复次数的元字符（如`?`或`*`或`{2}`，等等）只作用于紧挨着它的前一个字符或元字符。

我们来看一个例子。有些短语（例如Windows 2000）虽然由多个单词构成，但其实是一个整体。有许多HTML程序员喜欢让这类短语在浏览器里显示在同一行上。为了确保这一点，他们会在编写HTML文档时在这些短语的单词之间使用非换行型空格（` `，`nbsp`是“non-breaking space”的缩写，其含义是“不是换行符的空格”）而不是普通的空格。下面就是一个这样的例子：

文本

```
Hello, my name is Ben Forta, and I am
the author of books on SQL, ColdFusion, WAP,
Windows  2000, and other subjects.
```



正则表达式

```
 {2,}
```

66

结果

```
Hello, my name is Ben Forta, and I am
the author of books on SQL, ColdFusion, WAP,
Windows  2000, and other subjects.
```

分析

 是HTML语言中的非换行空格字符。在这里使用模式 {2,}的本意是希望它能把 连续两次或更多次的重复出现找出来，但它没能给出我们所预期的结果。为什么会这样？因为{2,}只作用于紧挨着它的前一个字符——那是一个分号。如此一来，这个模式只能匹配像 ;;;;这样的文本，但无法匹配 。

7.2 子表达式

这就引出了子表达式的概念。子表达式是一个更大的表达式的一部分；把一个表达式划分为一系列子表达式的目的是为了把那些子表达式当作一个独立元素来使用。子表达式必须用(和)括起来。



提示 (和)是元字符。如果需要匹配(和)本身，就必须使用它的转义序列\(和\)。

为了演示子表达式的用法，我们来看看刚才的那个例子：

文本

```
Hello, my name is Ben Forta, and I am
the author of books on SQL, ColdFusion, WAP,
Windows  2000, and other subjects.
```

正则表达式

```
( ){2,}
```

67

结果

```
Hello, my name is Ben Forta, and I am
the author of books on SQL, ColdFusion, WAP,
Windows  2000, and other subjects.
```

分析

`( )`是一个子表达式，它将被视为一个独立元素，而紧跟在它后面的`{2,}`将作用于这个子表达式（不仅仅是分号）。这个模式解决了我们的问题。

我们再来看一个例子，这次是用一个正则表达式来查找IP地址。IP地址的格式是以英文句号分隔的四组数字，例如12.159.46.200。因为每组数字由1个、2个或3个数字字符构成，所以这4组数字可以统一使用模式`\d{1,3}`来匹配。下面就是这个例子：

文本

```
Pinging hog.forta.com [12.159.46.200]
with 32 bytes of data:
```

正则表达式

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

结果

```
Pinging hog.forta.com [12.159.46.200]
with 32 bytes of data:
```

分析

`\d{1,3}`在这个模式里重复了4次，它们分别匹配IP地址里的一组数字。IP地址里的4组数字由`.`分隔，该字符由模式里的转义序列`\.`负责匹配。

稍微留意一下就会发现，在这个例子里，模式`\d{1,3}\.`（最多3个数字字符、后面跟着一个`.`）连续出现了3次，它同样可以被表达为一个重复。下面是这个例子的另一种解决方案：

文本

```
Pinging hog.forta.com [12.159.46.200]
with 32 bytes of data:
```

正则表达式

```
(\d{1,3}\.){3}\d{1,3}
```



结果

```
Pinging hog.forta.com [12.159.46.200]
with 32 bytes of data:
```

分析

这个模式与前面那个有着同样的效果，但我们这次使用了另一种语法：先用(和)把表达式\d{1,3}\.括起来使它成为一个子表达式，再用\d{1, 3}\ .){3}把这个子表达式重复了3次（它们对应着IP地址里的前3组数字），最后面的\d{1, 3}用来匹配IP地址里的最后一组数字。



注意 在上面这个例子里，使用(\d{1, 3}\ .){4}作为模式是不妥当的。你能分析出为什么不能用它来解决这个问题吗？



提示 为了提高可读性，有不少用户喜欢给表达式的每一个子表达式都加上括号。比如，把上面那个例子里的模式写成(\d{1, 3}\ .){3}(\d{1, 3})。这种做法在语法上完全成立，对表达式的实际行为也没有任何不良影响（但视乎具体的正则表达式实现，这对匹配操作的速度可能会有点儿影响）。

子表达式是一个非常重要的概念，所以我们认为有必要再给大家看一个例子，它不涉及重复次数问题。在下面的例子里，我们的任务是把一条用户记录里的年份数字完整地匹配出来：

文本

```
ID: 042
SEX: M
DOB: 1967-08-17
Status: Active
```

正则表达式

```
19|20\d{2}
```

结果

```
ID: 042
SEX: M
DOB: 1967-08-17
Status: Active
```

分析

这个例子需要我们构造一个模式去查找一个4位数的年份数字。为了排除没有实际意义的结果，我们把前两位数字限定为19和20。这个模式里的|字符是正则表达式语言里的或操作符，19|20将匹配数字序列19或20。既然如此，模式19|20\d{2}应该匹配以19或20开头的四位数字（19或20的后面再跟着两位数字）。可是，这个模式的匹配结果与我们的预期并不相符，它只匹配到了19，随后两位数字没有被匹配到。为什么会这样？因为|操作符是把位于它左边和右边的两个部分都作为一个整体来看待的，它会把模式19|20\d{2}解释为19或20\d{2}（也就是把\d{2}解释为以20开头的那个表达式的一部分）。换句话说，它将匹配数字序列19或以20开头的任意4位数字。最终的结果你们已经看到了，它只匹配到了19。

这个例子的正确答案是把19|20归为一个子表达式，如下所示：

文本

```
ID: 042
SEX: M
DOB: 1967-08-17
Status: Active
```

正则表达式

```
(19|20)\d{2}
```

结果

```
ID: 042
SEX: M
DOB: 1967-08-17
Status: Active
```

70

分析

我们把所有的选项都归纳到了一个子表达式里，这将向|表明我们打

算匹配的是这个子表达式里的选项之一。`(19|20)\d{2}`正确地匹配到了1967；当然，以19或20开头的任何一个4位数字都将与这个模式相匹配。今后（比如，从现在算起100年内），如果需要修改这段代码以包括以21开头的年份，只要把这个模式改成`(19|20|21)\d{2}`就可以了。

本章讨论的只是子表达式的用途之一。子表达式还有另外一个非常重要的用途，我们将在第8章里对之进行讨论。

7.3　子表达式的嵌套

子表达式允许嵌套。事实上，子表达式允许多重嵌套，这种嵌套的层次在理论上没有限制，但在实际工作中还是应该遵循适可而止的原则。

多重嵌套的子表达式可以构造出功能极其强大的正则表达式来，但那难免会让模式变得难以阅读和理解，而这也正是很多人觉得正则表达式难以学习和掌握的原因之一。这种表面现象掩盖了这样一个事实：绝大多数嵌套子表达式都没有它们看上去那么复杂。

为了演示嵌套子表达式的用法，我们再去看看刚才那个匹配IP地址的例子。下面是我们刚才使用的模式（先是一个连续重复3次的子表达式，然后是最后一组数字）：

正则表达式

```
(\d{1,3}\.){3}\d{1,3}
```

这个模式有什么不对的地方吗？从语法上讲，它完全正确。IP地址由四组数字构成，每组数字由1到3个数字字符构成，它们之间以英文句号分隔。说这个模式正确，是因为所有合法的IP地址都与之相匹配。但深入研究一下就会发现，这个模式还可以匹配其他一些东西；说得明白点儿，不合法的IP地址也能与之相匹配。

IP地址由4个字节构成，IP地址中的4组数字分别对应着那4个字节，所以IP地址里的每组数字的取值范围也就是单个字节的表示范围，即0~255。这意味着IP地址里的每一组数字都不能大于255，可是上面那个模式还能匹配诸如345、700、999之类的数字序列，而这些数字在IP地址里都是非法的。

71



> **注意**　有句话希望你能牢牢记住：把必须匹配的情况考虑周全并写出一个匹配结果符合预期的正则表达式很容易，但把不需要匹配的情况也考虑周全并确保它们都将被排除在匹配结果以外往往要困难得多。

如果有办法设定各种取值范围的话，事情会简单得多，但可惜的是正则表达式只是一种工具，而且还是一种不懂数学运算的工具，它们在匹配字符的时候并不真正关心那些字符到底是什么以及有什么含义。你的数学能力再好在这里也帮不上忙。

真的没有解决这个问题的办法吗？未必，只要你们能够充分发挥你们的逻辑思维能力，就能解决与正则表达式有关的任何难题。这里的基本思路是：在构造一个正则表达式的时候，一定要把你想匹配什么和你不想匹配什么详尽地定义清楚。下面是一个合法的IP地址里的各组数字必须且只能符合的规则，我们随后将根据这些规则来构造一个相应的模式：

- 任何一个1位或2位数字。
- 任何一个以1开头的3位数字。
- 任何一个以2开头、第2位数字在0~4之间的3位数字。
- 任何一个以25开头、第3位数字在0~5之间的3位数字。

像这样把所有的正则全部罗列出来之后，构造一个同时符合所有原则的模式的具体步骤也就清晰了。下面是这个例子的继续：

文本

```
Pinging hog.forta.com [12.159.46.200]
with 32 bytes of data:
```

正则表达式

```
(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}
➥((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))
```

结果

```
Pinging hog.forta.com [12.159.46.200]
with 32 bytes of data:
```

分析

这个模式的使用效果显而易见，但它还是需要仔细研读才能看明白。这个模式由一系列嵌套子表达式构成。我们先来说说由4个子表达式构成的(((\d{1, 2}) | (1\d{2}) | (2[0-4]\d) | (25[0-5])\ .)：(\d{1, 2})匹配任意一位或两位数字（0~99）；(1\d{2})匹配以1开头的任意三位数字（100~199）；(2[0-4]\d)匹配整数200~249；(25[0-5])匹配整数250~255。这几个子表达式通过|操作符结合为一个更大的子表达式（其含义是只需匹配这4个子表达式之一即可）。随后的\ .用来匹配.字符，它与前4个子表达式构成的子表达式又构成了一个更大的子表达式（4组数字选项和\ .），而接下来的{3}表明需要重复3次。最后，数值范围又重复了一次（这次省略了尾部的\ .）以匹配IP地址里的最后一组数字。通过把4组数字的取值范围都限制在0~255之间，这个模式准确无误地做到了只匹配合法的IP地址、不匹配非法的IP地址。



提示 上面这个例子里的正则表达式看起来很难理解。把它们弄明白的关键是要把它们分解开、每次只分析和理解一个子表达式。在分析各个子表达式的时候，应该按照先内后外的原则来进行而不是从第一个字符开始一个字符一个字符地去尝试。你有过几次这样的经验之后就会发现，嵌套子表达式并不像它们看上去那么复杂。

7.4 小结

子表达式的作用是把同一个表达式的各个相关部分组合在一起。子表达式必须用(和)来定义。子表达式的常见用途包括：对重复次数元字符的作用对象做出精确的设定和控制、对|操作符的OR条件做出准确的定义，等等。如有必要，子表达式还允许嵌套使用。

第8章

回溯引用：前后一致匹配



第7章介绍了子表达式的基本用途之一：把一组字符编组为一个字符集合。这样的字符集合主要用于精确设定需要重复匹配的文本及其重复次数。本章将讨论子表达式的另一个重要用途——定义回溯引用（backreference）。

8.1 回溯引用有什么用

为了理解回溯引用的概念，我们最好是看一个例子。HTML程序员经常使用标题标签（<H1>到<H6>，以及配对的结束标签）来定义和排版Web页面里的标题文字。现在，我们不妨假设你需要把某个Web页面里的所有标题文字全都查找出来，而不管它的级别是多少。下面就是这个例子：

文本

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
</BODY>
```

正则表达式

```
<[hH]1>.*</[hH]1>
```

结果

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
</BODY>
```

分析

模式`<[hH]1>.*</[hH]1>`只能匹配一级标题（从`<H1>`或`<h1>`到`</H1>`或`</h1>`；HTML语言不区分字母的大小写）。但我们刚才说的是匹配任意级别的标题（HTML文档里的标题总共有6个级别），这应该怎么办呢？

最容易想到的办法是用一个字符集合来代替1，如下所示：

文本

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
</BODY>
```

正则表达式

```
<[hH][1-6]>.*?</[hH][1-6]>
```

结果

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
</BODY>
```

分析

这个模式看来不错，`<[hH][1-6]>`匹配任何一级标题的开始标签（具

体到这个例子，它匹配到了<H1>和<H2>)，</[hH][1-6]>匹配任何一级
75
标题的结束标签（具体到这个例子，它匹配到了</H1>和</H2>)。



注意　这里使用的是.*?（懒惰型）而不是.*（贪婪型）。我们在第5章里讲过，*和其他几个元字符是“贪婪型”元字符，所以模式<[hH][1-6]>.*</[hH][1-6]>有可能会从第2行的<H1>一直匹配到第6行的</H2>，这可不是我们想要的结果；使用“懒惰型”元字符.*?解决了这个问题。
之所以说“有可能”而不是“肯定”，是因为在这个特定的例子里即便是使用了“贪婪型”元字符也不一定会有问题。一般来说，元字符.不匹配换行符，而上例中的每个标题都各自占据一行。但在这里使用懒惰型元字符没有任何坏处——事前小心总比事后后悔好。

现在成功了吗？未必。看看下面这个例子（这次使用的是还是刚才那个模式），你就知道我为什么这样说了：

文本

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

正则表达式

```
<[hH][1-6]>.*?</[hH][1-6]>
```

结果

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
```

76



```
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

分析

在这个例子里，原始文本里有一个标题是以<H2>开头、以<H3>结束的。这显然是一个不合法的标题，但它与我们所使用的模式匹配上了。

出现这种情况的根源是这个模式的第2部分（用来匹配结束标签的那个部分）对这个模式的第1部分（用来匹配开始标签的那个部分）毫无所知。要想彻底解决这个问题，就只能求助于回溯引用。

8.2 回溯引用匹配

我们等会儿再去解决匹配HTML标题的问题。先来看一个比较简单的例子，这个问题如果不使用回溯引用将根本无法解决。

假设你有一段文本，你想把这段文本里所有连续重复出现的单词（打字错误，其中有一个单词输了两遍）找出来。显然，在搜索某个单词的第二次出现时，这个单词必须是已知的。回溯引用允许正则表达式模式引用前面的匹配结果（具体到这个例子，就是前面匹配到的单词）。

把这个问题弄明白的最佳办法是看看它到底是如何工作的。下面是一段包含着3组重复单词的文本，它们就是我们要找的东西：

文本

```
This is a block of of text,
several words here are are
repeated, and and they
should not be.
```

正则表达式

```
[ ]+(\w+)[ ]+\1
```

结果

```
This is a block of of text,
several words here are are
repeated, and and they
should not be.
```

分析

这个模式找到了我们想要的东西，但它是如何做到这一点的呢？[ ]+匹配一个或多个空格，\w+匹配一个或多个字母数字字符，[ ]+匹配随后的空格。注意，\w+是括在括号里的，它是一个子表达式。这个子表达式不是用来进行重复匹配的，这里根本不涉及重复匹配的问题。这个子表达式只是把整个模式的一部分单独划分出来以便在后面引用。这个模式的最后一部分是\1；这是一个回溯引用，而它引用的正是前面划分出来的那个子表达式：当(\w+)匹配到单词of的时候，\1也匹配单词of；当(\w+)匹配到单词and的时候，\1也匹配单词and。



注意　回溯引用指的是模式的后半部分引用在前半部分中定义的子表达式（如上例所示）。

\1到底代表着什么？它代表着模式里的第1个子表达式，\2代表着第2个子表达式、\3代表着第3个；依次类推。于是，在上面那个例子里，[ ]+(\w+)[ ]+\1将匹配同一个单词的连续两次重复出现。



提示　你们可以把回溯引用想像成变量。

看过回溯引用的用法之后，我们再回过头来看看应该如何解决匹配HTML标题的问题。利用回溯引用，构造一个模式去匹配任何一级标题的开始标签和与之配对的结束标签（忽略任何不配对的标签组合）对我们来说已经不是什么难题了。下面就是这个例子：

文本

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

正则表达式

```
<[hH]([1-6])>.*?</[hH]\1>
```

结果

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

分析

总共找到了3个匹配：1个一级标题（`<H1>...</H1>`）和2个二级标题（`<H2>...</H2>`）。`<[hH]([1-6])>`匹配任何一级标题的开始标签，但我们这次用`(`和`)`把`[1-6]`括了起来，使它成为了一个子表达式。这样一来，我们就可以在用来匹配标题结束标签的`</[hH]\1>`用`\1`来引用这个子表达式了。子表达式`([1-6])`匹配数字1~6，`\1`只匹配与之相同的数字。这样一来，原始文本里的`<H2>This is not valid HTML</H3>`就不会被匹配到了。

79



注意 不同的正则表达式在实现回溯引用的语法方面往往有着巨大的差异。

JavaScript使用`\`来标识回溯引用（`\`与`$`配合进行替换操作时是例外），Macromedia ColdFusion和vi也是如此。.NET正则表达式将返回一个对象，该对象的Groups属性包含着所有的匹配——如果你使用的是C#语言，`match.Groups[1]`对应着第一个匹配；如果你使用的是Visual Basic .NET，`match.Groups(1)`对应着第一个匹配。PHP把这些信息返回为一个名为`$matches`数组，`$matches[1]`对应着第1个匹配（但这一行为会根据你在匹配操作中具体使用的命令选项发生变化）。Java和Python将返回一个包含着一个名为`group`的数组的匹配对象。

有关细节参阅附录A。



警告　回溯引用只能用来引用模式里的子表达式(用(和)括起来的正则表达式片段)。



提示　回溯引用匹配通常从1开始计数（\1、\2，等等）。在许多实现里，第0个匹配（\0）可以用来代表整个正则表达式。



注意　正如看到的那样，子表达式是通过它们的相对位置来引用的：\1对应着第1个子表达式，\5对应着第5个子表达式，等等。虽然受到普遍的支持，但这种语法存在着一个严重的不足：如果子表达式的相对位置发生了变化，整个模式也许就不能再完成原来的工作，删除或添加子表达式的后果可能更为严重。

为了弥补这一不足，一些比较新的正则表达式实现还支持“命名捕获”(named capture)：给某个子表达式起一个唯一的名字，然后用这个名字（而不是相对位置）来引用这个子表达式。因为命名捕获还没有得到广泛支持，而且已支持的实现具体的语法也极不统一，所以本书没有对此进行讨论。但是，如果你正在使用的正则表达式实现支持命名捕获功能（如.NET），你应该充分利用。

80

8.3　回溯引用在替换操作中的应用

到目前为止，你们在这本书里见到的正则表达式都是用来执行搜索的，即在一段文本里查找特定的内容。你在今后的实际工作中也会发现，你所编写的绝大多数正则表达式模式也可以用来搜索文本。但这并不是正则表达式的全部功能；正则表达式还可以用来完成各种复杂的替换操作。

简单的文本替换操作无须使用正则表达式就可以完成。比如说，如果只是把某个文档里的CA全部替换为California或把MI全部替换为

Michigan的话，用正则表达式来完成这些替换就未免有点儿大材小用了。这句话的意思并不是说正则表达式不能用来执行这种替换，只是那么做没有什么实际价值。事实上，用普通的字符串处理函数来完成这种替换反而会更容易一些。

正则表达式更适用于复杂的替换，尤其是需要使用回溯引用的场合，那才能体现出正则表达式的真正威力。下面是一个我们在第5章里见过的例子：

文本

```
Hello, ben@forta.com is my email address.
```

正则表达式

```
\w+[\w\.]*@[\w\.]+\.\w+
```

结果

Hello, **ben@forta.com** is my email address.

分析

这个模式可以把原始文本里的电子邮件地址查找出来（详细分析见第5章）。

现在，假设你需要把原始文本里的电子邮件地址全都转换为可点击的链接，你该怎么办？在HTML文档里，你需要使用<A HREF="maito:user@address.com">user@address.com</A>这样的语法来创建一个可点击的电子邮件地址。能不能用一个正则表达式把一个电子邮件地址转换为这种可点击的地址格式呢？当然能，而且非常容易——但前提是你得使用回溯引用，如下所示：

81

文本

```
Hello, ben@forta.com is my email address.
```

正则表达式

```
(\w+[\w\.]*@[\w\.]+\.\w+)
```

替换

```
<A HREF="mailto:$1">$1</A>
```



结果

```
Hello, <A HREF="mailto:ben@forta.com">ben@forta.com</A>
is my email address.
```

分析

替换操作需要用到两个正则表达式：一个用来给出搜索模式，另一个用来给出匹配文本的替换模式。回溯引用可以跨模式使用，在第一个模式里被匹配的子表达式可以用在第二个模式里。这里使用的模式`(\w+[\w\ .]*@[\w\ .]+\ .\w+)`与我们以前使用的完全一样（匹配电子邮件地址），但这次把它写成了一个子表达式。这样一来，被匹配到的文本就可以用在替换模式里了。`<A HREF="mailto:$1">$1</A>`使用了两次被匹配的子表达式：一次是在HREF属性里（来定义`mailto:...`），另一次是做为可点击文本。具体到这个例子，`ben@forta.com`变成了`<A HREF="mailto:ben@forta.com"> ben@forta.com </A>`，而这正是我们想要的结果。



警告　我们刚才讲过，回溯引用语法在不同的正则表达式实现里有很大的差异：JavaScript用户需要用$来代替\；ColdFusion用户在查找和替换操作里都必须使用\。



提示　正如你在上面这个例子里看到的那样，同一个子表达式可以被引用任意多次——只要在需要用到它的地方写出它的回溯引用就行了。

82

我们再来看一个例子。在一个用来保存用户信息的数据库里，电话号码被保存为313-555-1234。现在，你需要把电话号码重新排版为(313) 555-1234。下面就是这个例子：

文本

```
313-555-1234
248-555-9999
810-555-9000
```

正则表达式

```
(\d{3})(-)(\d{3})(-)(\d{4})
```

替换

```
($1) $3-$5
```

结果

```
(313) 555-1234
(248) 555-9999
(810) 555-9000
```

分析

和刚才一样，这里也使用了两个正则表达式模式。第1个模式看起来很复杂，我们来分析一下。`(\d{3})(-)(\d{3})(-)(\d{4})`用来匹配一个电话号码，它被划分为5个子表达式（5个组成部分）：第1个子表达式`(\d{3})`匹配前3位数字，第2个子表达式`(-)`匹配-字符，等等。最终的结果是一个电话号码被划分成了5个部分（每个部分分别对应着一个子表达式）：区号、一个连字符、电话号码的前3位数字、又一个连字符、电话号码的后4位数字。这5个部分都可以单独拿出来使用，负责重新排版电话号码的替换模式`($1) $3-$5`只用到了它们当中的3个，剩下的两个没有用到，但这已足以把`313-555-1234`转换为`(313) 555-1234`。



提示 在对文本进行重新排版的时候，把文本分解成多个子表达式的做法往往非常有用，这可以让我们对文本的排版效果做出更精确的控制。

83

大小写转换

有些正则表达式实现允许我们使用表8-1列出的元字符对字母进行大小写转换。

表8-1　用来进行大小写转换的元字符

元字符	说　明
\E	结束\L或\U转换
\l	把下一个字符转换为小写
\L	把\L到\E之间的字符全部转换为小写
\u	把下一个字符转换为大写
\U	把\U到\E之间的字符全部转换为大写

\l和\u只能把下一个字符（或子表达式）转换为小写或大写。\L和\U将把它后面的所有字符转换为小写或大写，直到遇上\E为止。

下面是一个简单的例子，把一级标题（<H1>...</H1>）的标题文字转换为大写：

文本

```
<BODY>
<H1>Welcome to my Homepage</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```

正则表达式

```
(<[Hh]1>)(.*?)(</[Hh]1>)
```

替换

84

```
$1\U$2\E$3
```

结果

```
<BODY>
<H1>WELCOME TO MY HOMEPAGE</H1>
Content is divided into two sections:<BR>
<H2>ColdFusion</H2>
Information about Macromedia ColdFusion.
<H2>Wireless</H2>
Information about Bluetooth, 802.11, and more.
<H2>This is not valid HTML</H3>
</BODY>
```



分析

模式 (<[Hh]1>)(.*?)(</[Hh]1>) 把一级标题分成了3个子表达式：开始标签、标题文字、结束标签。第2个模式再把文本重新组合起来：**$1**包含着开始标签，**\U$2\E**把第2个子表达式（标题文字）转换为大写，**$3**包含着结束标签。

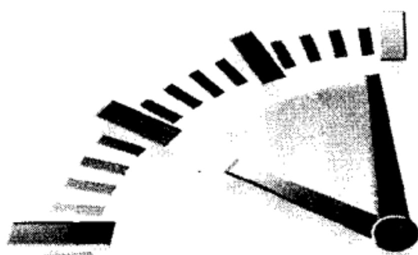
8.4 小结

子表达式用来定义字符或表达式的集合。除了可以用在重复匹配操作中以外（详见第7章），子表达式还可以在模式的内部被引用。这种引用被称为回溯引用。回溯引用的语法在不同的正则表达式实现里有很大的差异。回溯引用在文本匹配和文本替换操作里非常有用。

85



第 9 章



前后查找

到目前为止，我们见过的正则表达式都是用来匹配文本的，但有时我们还需要用正则表达式标记要匹配的文本的位置（而不仅仅是文本本身）。这就引出了前后查找（lookaround，对某一位置的前、后内容进行查找）的概念，我们将在这一章对此做专题讨论。

9.1 前后查找

我们还是先来看一个例子：你要把一个Web页面的页面标题提取出来。HTML页面标题是出现在`<TITLE>`和`</TITLE>`标签之间的文字，而这对标签又必须嵌在HTML代码的`<HEAD>`部分里。下面就是这个例子：

文本

```
<HEAD>
<TITLE>Ben Forta's Homepage</TITLE>
</HEAD>
```

正则表达式

```
<[tT][iI][tT][lL][eE]>.*</[tT][iI][tT][lL][eE]>
```

结果

```
<HEAD>
<TITLE>Ben Forta's Homepage</TITLE>
</HEAD>
```

分析

`<[tT][iI][tT][lL][eE]>.*</[tT][iI][tT][lL][eE]>`匹配`<TITLE>`标签（大写、小写或大小写混用）、`</TITLE>`标签以及这两个

标签之间的任何文字。这个模式的效果与我们的预期基本相符，但不够理想。

86

为什么这么说呢？因为只有页面标题才是我们需要的，而我们找到的匹配里还包含着<TITLE>和</TITLE>标签。能不能只返回页面标题的文字部分呢？

办法之一是使用子表达式（参见第7章）。我们可以利用子表达式把被匹配文本划分为3个部分：开始标签、标题文字、结束标签。把被匹配文本划分为多个部分之后，从它们当中提取且只提取出我们需要的东西就很容易了。

可是，明知是自己并不真正需要的东西（比如上例中的<TITLE>和</TITLE>标签），还把它们检索出来岂不是毫无意义。“先想办法把它们检索出来、再以手动方式排除它们”这既浪费时间，又容易招致不必要的后患。在遇到这类问题的时候，你真正需要的是这样一个模式，它包含的匹配本身并不返回，而是用于确定正确的匹配位置，它并不是匹配结果的一部分。换句话说，你需要进行“前后查找”[①]。



注意 本章将对向前查找(lookahead)和向后查找(lookbehind)都进行讨论。常见的正则表达式实现都支持前者，但支持后者的就没那么多了。

Java、.NET、PHP和Perl都支持向后查找（但有一些限制），JavaScript和ColdFusion不支持向后查找。

9.2 向前查找

向前查找指定了一个必须匹配但不在结果中返回的模式。向前查找实际就是一个子表达式，而且从格式上看也确实如此。从语法上看，一个向前查找模式其实就是一个以?=开头的子表达式，需要匹配的文本跟在=的后面。

① 前后查找中的前、后指模式与被查找文本的相对位置而言，左为前。——编者注



提示 有些正则表达式文档使用术语“消费”（consume）来表述“匹配和返回文本”的含义。在向前查找里，被匹配的文本不包含在最终返回的匹配结果里，这被称为“不消费”。

87

我们来看一个例子。例子里的原始文本是一些URL地址，而你的任务是把它们的协议名部分提取出来（为下一步处理做准备）。下面就是这个例子：

文本

```
http://www.forta.com/
https://mail.forta.com/
ftp://ftp.forta.com/
```

正则表达式

```
.+(?=:)
```

结果

```
http://www.forta.com/
https://mail.forta.com/
ftp://ftp.forta.com/
```

分析

在上面列出的URL地址里，协议名与主机名之间以一个:分隔。模式.+匹配任意文本（第1个匹配是http），子表达式(?=:)匹配:。注意，被匹配到的:并没有出现在最终的匹配结果里；我们用?=向正则表达式引擎表明：只要找到:就行了，不要把它包括在最终的匹配结果里——用术语来说，就是“不消费”它。

为了更好地理解?=的作用，我们再来看一个同样的例子，但这次不使用向前查找元字符：

文本

```
http://www.forta.com/
https://mail.forta.com/
ftp://ftp.forta.com/
```

正则表达式

```
.+(:)
```

结果

```
http://www.forta.com/
https://mail.forta.com/
ftp://ftp.forta.com/
```

88

分析

子表达式(:)正确地匹配到了:并消费了这个字符——:出现在了最终的匹配结果里。

这两个例子的区别是前一个用来匹配:的模式是(?=:)，后一个用来匹配:的模式是(:)。这两个模式所匹配的东西是一样的——都是紧跟在协议名后面的那个:，它们之间的区别只是被匹配到的:字符有没有出现在最终的匹配结果里而已。在使用向前查找的时候，正则表达式分析器将向前查找并处理:匹配，但不会把它包括在最终的搜索结果里。模式.+(:)查找到并且匹配结果包含:，模式.+(?=:)查找到但匹配结果不包含:。



注意 向前查找(和向后查找)匹配本身其实是有返回结果的，只是这个结果的字节长度永远是0而已。因此，前后查找操作有时也被称为零宽度（zero-width）匹配操作。



提示 任何一个子表达式都可以转换为一个向前查找表达式，只要给它加上一个?=前缀即可。在同一个搜索模式里可以使用多个向前查找表达式，它们可以出现在模式里的任意位置（而不仅仅是出现在整个模式的开头——就像你们在上面看到的那样）。

9.3 向后查找

正如你刚看到的那样，?=将向前查找（查找出现在被匹配文本之后的字符，但不消费那个字符）。因此，?=被称为向前查找操作符。除了向前查找，许多正则表达式实现还支持向后查找，也就是查找出现在被匹配文本之前的字符（但不消费它），向后查找操作符是?<=。

89



提示　分不清?=、?<=与其他?的话，有个简单的办法可以帮你分辨它们：有小于号的是向后查找操作符——你可以把这个小于号想像成一个箭头，它指向文本阅读方向的后方[①]。

?<=与?=的具体使用方法大同小异；它必须用在一个子表达式里，而且后跟要匹配的文本。

下面是一个例子。你从某个数据库里搜索出了一份产品目录，但你只需要把那些产品的价格提取出来：

文本

```
ABC01: $23.45
HGG42: $5.31
CFMX1: $899.00
XTC99: $69.96
Total items found: 4
```

正则表达式

```
\$[0-9.]+
```

结果

```
ABC01: $23.45
HGG42: $5.31
CFMX1: $899.00
XTC99: $69.96
Total items found: 4
```

分析

\$匹配$，[0-9.]+匹配价格。

如上所示的匹配结果符合你的预期。但如果你不想让$出现在最终的匹配结果里，你该怎么办？从这个模式里简单地把$去掉能达到目的吗？

文本

```
ABC01: $23.45
HGG42: $5.31
CFMX1: $899.00
XTC99: $69.96
Total items found: 4
```


① 因为要匹配文本相对于模式的方向（对应“向前查找”的“前”）与文本阅读方向正相反，记忆向后查找<号的方向容易引起误解，可以直接将“？<=”读成“向……之后查找”。——编者注

正则表达式

```
[0-9.]+
```

结果

```
ABC01: $23.45
HGG42: $5.31
CFMX1: $899.00
XTC99: $69.96
Total items found: 4
```

分析

这显然不是你想要的结果。你需要\$来确定应该匹配哪些文本，你只是不想让$出现在最终的匹配结果里而已。

怎么办？好办，这正是向后查找可以大显身手的地方，如下所示：

文本

op

正则表达式

```
(?<=\$)[0-9.]+
```

结果

```
ABC01: $23.45
HGG42: $5.31
CFMX1: $899.00
XTC99: $69.96
Total items found: 4
```

分析

问题迎刃而解了。`(?<=\$)`只匹配$，但不消费它；最终的匹配结果里只有价格数字（没有前缀的$字符）。

我们来对比一下这个例子的第一个和最后一个表达式：`\$[0-9.]+`匹配一个$字符和一个美元金额；`(?<=\$)[0-9.]+`也匹配一个$字符和一个美元金额。这两个模式所查找的东西是一样的，它们之间的区别只体现在它们的匹配结果里。前一个模式的匹配结果包含着$，后一个模式的匹配结果不包含$字符，虽然它必须通过匹配$字符才能正确地找到那些价格数字。

91



警告　向前查找模式的长度是可变的，它们可以包含.和+之类的元字符，所以它们非常灵活。

而向后查找模式只能是固定长度——这是一条几乎所有的正则表达式实现都遵守的限制。

9.4　把向前查找和向后查找结合起来

向前查找和向后查找可以组合在一起使用，就像下面这个例子所演示的那样（这个例子解决了我们在本章刚开始时提出的问题）：

文本

```
<HEAD>
<TITLE>Ben Forta's Homepage</TITLE>
</HEAD>
```

正则表达式

```
(?<=<[tT][iI][tT][lL][eE]>).*(?=</[tT][iI][tT][lL][eE]>)
```

结果

```
<HEAD>
<TITLE>Ben Forta's Homepage</TITLE>
</HEAD>
```

分析

问题解决了。(?<=<[tT][iI][tT][lL][eE]>)是一个向后查找操作，它匹配（但不消费）<TITLE>；而(?=</[tT][iI][tT][lL][eE]>)是一个向前查找操作，它匹配（但不消费）</TITLE>。最终返回的匹配结果包含且仅包含标题文字（用术语来说，就是只有标题文字被消费了）。

92



提示　为减少歧义，在上面这个例子里，你应该对<（需要匹配的第一个字符）进行一下转义，也就是把(?<=<替换为(?<=\<。

9.5 对前后查找取非

到目前为止正如你看到的那样，向前查找和向后查找通常用来匹配文本，其目的是为了确定将被返回为匹配结果的文本的位置（通过指定匹配结果的前后必须是哪些文本）。这种用法被称为正向前查找（positive lookahead）和正向后查找（positive lookbehind）。术语“正”指的是寻找匹配的事实。

前后查找还有一种不太常见的用法叫做负前后查找（negative lookaround）[①]。负向前查找（negative lookahead）将向前查找不与给定模式相匹配的文本，负向后查找（negative lookbehind）将向后查找不与给定模式相匹配的文本。

我们在第3章曾经介绍过一个用来对字符集合进行取非处理的操作符^，但^不能用来对前后查找进行取非处理。这里必须使用另外一种语法：前后查找必须用!来取非（它将替换掉=）。表9-1列出了所有的前后查找操作符。

表9-1 各种前后查找操作符

操作符	说 明
(?=)	正向前查找
(?!)	负向前查找
(?<=)	正向后查找
(?<!)	负向后查找

93



提示 一般来说，凡是支持向前查找的正则表达式实现都同时支持正向前查找和负向前查找。类似地，凡是支持向后查找的正则表达式实现都同时支持正向后查找和负向后查找。

为了演示正向后查找和负向后查找之间的区别，我们来看一个例子。下面是一段包含着一些数值的文本，其中既有价格又有数量。我们先来查找且只查找价格：

---

① 此处的“负”译为“取非”更好理解。——编者注

文本

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

正则表达式

```
(?<=\$)\d+
```

结果

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

分析

这与我们刚见过的例子非常相似。\d+匹配数值（一个或多个数字字符），(?<=\$)向后查找（但不消费）字符$（这个字符在模式里被转义为\$）。这个模式正确地匹配到了两个用来表示价格的数值，那些用来表示数量的数字没有出现在最终的匹配结果里。

接下来，我们再去查找且只查找数量：

文本

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

正则表达式

94

```
\b(?<!\$)\d+\b
```

结果

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

分析

\d+还是匹配数值，但这次只匹配数量，不匹配价格。表达式(?<!\$)是一个负向后查找，它使得最终的匹配结果只包含那些不以$开头的数值。把操作符?<=改为操作符?<!使得整个模式从一个正向后查找变成了一个负向后查找。

细心的读者可能已经注意到了，在上面这个例子里，我们还在那个负向后查找模式里用\b元字符定义了两个单词边界。我们为什么要那么做呢？你看过下面这个没有使用单词边界的例子里就明白了。

文本

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

正则表达式

```
(?<!\$)\d+
```

结果

```
I paid $30 for 100 apples,
50 oranges, and 60 pears.
I saved $5 on this order.
```

分析

请看，因为没有使用单词边界，**$30**里的**0**也出现在了最终的匹配结果里。这是因为那个**0**字符的前一个字符是3而不是$，它完全符合模式(?<!\$)\d+的匹配要求。把这个模式用\b括起来从根本上解决了这个问题。

9.6 小结

有了向后查找，我们就可以对最终的匹配结果包含且只包含哪些内容做出更精确的控制。前后查找操作使我们可以利用子表达式来指定文本匹配操作的发生位置，并收到只匹配不消费的效果。正向前查找要用(?=)来定义，负向前查找要用(?!)来定义。有些正则表达式实现还支持正向后查找（相应的操作符是(?<=)）和负向后查找（相应的操作符是(?<!)）。

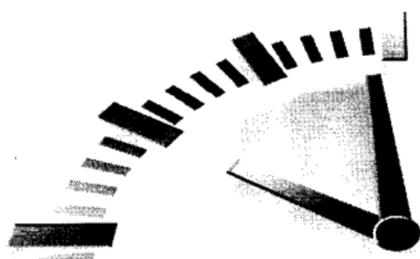
95

96



第 10 章

嵌入条件



正则表达式语言还有一种威力强大（但不经常被用到）的功能——在表达式的内部嵌入条件处理功能。本章将对此做专题讨论。

10.1 为什么要嵌入条件

(123)456-7890和123-456-7890都是可接受的北美电话号码格式，而1234567890、(123)-456-7890和(123-456-7890)虽然都包含着数目正确的数字字符，但它们的格式都不对。如果让你来编写一个正则表达式并让它只匹配可接受的格式，不匹配其他格式，你会怎么做？

这个问题看似简单，其实颇有难度。下面是最容易想到的解决方案：

文本

```
123-456-7890
(123)456-7890
(123)-456-7890
(123-456-7890
1234567890
123 456 7890
```

正则表达式

```
\(?\d{3}\)?-?\d{3}-\d{4}
```

结果

```
123-456-7890
(123)456-7890
(123)-456-7890
(123-456-7890
```



```
1234567890
123 456 7890
```

97

分析

\(?匹配一个可选的左括号——请注意，这里必须对(进行转义；\d{3}匹配前三位数字；\)?匹配一个可选的右括号；-?匹配一个可选的连字符；\d{3}-\d{4}匹配剩余的七位数字（中间用一个连字符分隔）。原始文本中的最后两行不与这个模式匹配，但第3行和第4行与之匹配——这是不正确的，第3行的)后面多了一个-，第4行少了一个配对的右括号)。

把\)?-?替换为[\]-]?可以排除第3行（字符）或-只能出现一个，不允许两个同时出现），但第4行还是无法排除。正确的模式应该只在电话代码里有一个左括号(的时候才去匹配)。更准确地说，应该是如果电话号码里有一个左括号(，我们的模式必须去匹配)；如果不是这样，它就必须去匹配-。这种模式如果不使用条件处理根本无法编写。



警告 并非所有的正则表达式实现都支持条件处理①。

10.2 正则表达式里的条件

正则表达式里的条件要用?来定义。事实上，你们已经见过几种非常特定的条件了：

- ?匹配前一个字符或表达式——如果它存在的话。
- ?=和?<=匹配前面或后面的文本——如果它存在的话。

嵌入条件语法也使用了?，这并没有什么让人感到吃惊的地方——因为嵌入条件不外乎以下两种情况：

- 根据一个回溯引用来进行条件处理。
- 根据一个前后查找来进行条件处理。

10.2.1 回溯引用条件

回溯引用条件只在一个前面的子表达式搜索取得成功的情况下才允许使用一个表达式。听起来很费解，我们还是用一个例子来说明好了：

98

① MySQL和Java 1.4均不支持。

你需要把一段文本里的<IMG>标签全都找出来；不仅如此，如果某个<IMG>标签是一个链接（被括在<A>和</A>标签之间）的话，你还要把整个链接标签匹配出来。

用来定义这种条件的语法是?(backreference)true-regex)，其中?表明这是一个条件，括号里的backreference是一个回溯引用，true-regex是一个只在backreference存在时才会被执行的子表达式。

请看下面这个例子：

文本

```
<!-- Nav bar -->
<TD>
<A HREF="/home"><IMG SRC="/images/home.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/search"><IMG SRC="/images/search.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/help"><IMG SRC="/images/help.gif"></A>
</TD>
```

正则表达式

```
(<[Aa]\s+[^>]+>\s*)?<[Ii][Mm][Gg]\s+[^>]+>(?(1)\s*</[Aa]>)
```

结果

```
<!-- Nav bar -->
<TD>
<A HREF="/home"><IMG SRC="/images/home.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/search"><IMG SRC="/images/search.gif"></A>
<IMG SRC="/images/spacer.gif">
<A HREF="/help"><IMG SRC="/images/help.gif"></A>
</TD>
```

分析

这个模式不解释是不容易看明白的。`(<[Aa]\s+[^>]+>\s*)?`将匹配一个<A>或<a>标签（以及<A>或<a>标签的任意属性），这个标签可有可无（因为这个子表达式的最后有一个?）。接下来，`<[Ii][Mm][Gg]\s+[^>]+>`匹配一个<IMG>（大小写均可）及其任意属性。`(?(1)\s*</[Aa]>)`是一个回溯引用条件——?(1)的含义是：如果第一个回溯引用（具体到本例，就是<A>标签）存在，则使用

\s*</[Aa]>继续进行匹配（换句话说，只有当前面的<A>标签匹配成功，才继续进行后面的匹配）。如果(1)存在，\s*</[Aa]>将匹配结束标签</A>之后出现的任意空白字符。



注意　?(1)检查第一个回溯引用是否存在。在条件里，回溯引用编号（本例中的1）不需要被转义。因此，?(1)是正确的，?(\1)不正确（但后者通常也能工作）。

我们刚才使用的模式只在给定条件得到满足时才执行一个表达式。条件还可以有否则表达式，否则表达式只在给定的回溯引用不存在（也就是条件没有得到满足）时才会被执行。用来定义这种条件的语法是?(backreference)true-regex|false-regex)，这个语法接受一个条件和两个将分别在这个条件得到满足和没有得到满足时执行的子表达式。

这个语法提供了电话号码问题的解决方案，如下所示：

文本

```
123-456-7890
(123)456-7890
(123)-456-7890
(123-456-7890
1234567890
123 456 7890
```

正则表达式

```
(\()?\d{3}(?(1)\)|-)\d{3}-\d{4}
```

结果

```
123-456-7890
(123)456-7890
(123)-456-7890
(123-456-7890
1234567890
123 456 7890
```

100

分析

从结果看，这个模式解决了问题，但它是如何解决问题的呢？和前面一样，(\()?也匹配一个可选的左括号，但我们这次把它用括号括起

来得到了一个子表达式。随后的\d{3}匹配一位数字的区号。(?(1)\)|-)是一个回溯引用条件，它将根据条件是否得到满足而去匹配)或-——如果(1)存在（也就是找到了一个左括号），\)必须被匹配；否则，-必须被匹配。这样一来，只有配对出现的括号才会被匹配；如果没有使用括号或括号不配对，电话号码中的区号和其余数字之间的-分隔符必须被匹配。



提示　嵌入了条件的模式看上去往往非常复杂，而这往往意味着调试工作会变得非常困难。如果别无选择，先对整个模式的各组成部分分别进行调试，再把它们拼装到一起，这通常是一种比较好的办法。

10.2.2　前后查找条件

前后查找条件只在一个向前查找或向后查找操作取得成功的情况下才允许一个表达式被使用。定义一个前后查找条件的语法与定义一个回溯引用条件的语法大同小异——只需把回溯引用（括号里的回溯引用编号）替换为一个完整的前后查找表达式就行了。



注意　对前后查找操作的详细讨论见第9章。

作为一个例子，请你思考一下怎样匹配美国的邮政编码（简称ZIP编码）。美国邮政编码有两种格式，一种是12345形式的ZIP格式，另一种是12345-6789形式的ZIP+4格式。只有ZIP+4格式才必须使用连字符来分隔前5位和后4位数字。下面是一种解决方法：

101

文本

```
11111
22222
33333-
44444-4444
```

正则表达式

```
\d{5}(-\d{4})?
```



结果

```
11111
22222
33333-
44444-4444
```

分析

`\d{5}`匹配前5位数字，`(-\d{4})?`匹配一个连字符和后4位数字（它们必须一起出现或一起不出现）。

现在，请考虑这样一个问题：如果你不想匹配那些格式不正确的ZIP编码，你该怎么办？比如说，在上面这个例子里，在第3行原始文本的末尾有一个不应该出现在那里的连字符，但这个号码还是出现在了最终的匹配结果里。怎样才能让这个格式不正确的ZIP编码不出现在最终的匹配结果里呢？

下面是用一个前后查找条件来解决这个问题的方法，虽然这个问题并非只有这一种解决办法。

文本

```
11111
22222
33333-
44444-4444
```

正则表达式

```
\d{5}(?(?=-)-\d{4})
```

结果

```
11111
22222
33333-
44444-4444
```

102

分析

`\d{5}`匹配前5位数字。接下来是一个`(?(?=-)-\d{4})`形式的向前查找条件。这个条件使用了`?=-`来匹配（但不消费）一个连字符，如果条件得到满足（那个连字符存在），`-\d{4}`将匹配那个连字符和随后的4位

数字。这样一来，**33333**-将被排除在最终的匹配结果之外（它有一个连字符，所以满足给定条件，但那个连字符的后面没有必须出现在那里的4位数字）。



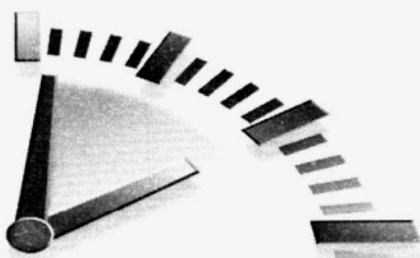
提示 在实际工作中，嵌入了前后查找条件的模式相当少见，这是因为我们往往可以用更简单的办法来达到同样的目的。

10.3 小结

在正则表达式模式里可以嵌入条件，只有当条件得到（或者没有得到）满足时，相应的表达式才会被执行。这种条件可以是一个回溯引用（含义是检查该回溯引用是否存在），也可以是一个前后查找操作。

103





附录A

常见应用软件和编程语言中的正则表达式

不同的正则表达式实现在基本的语法方面大都是一致的，但在正则表达式的具体用法方面往往有所不同。支持正则表达式的编程语言和应用软件各用各的调用方法，在许多细节上都有自己的一套方法。本附录将对比较流行的应用软件和编程语言中的正则表达式的用法和一些具体的注意事项进行描述。



注意 本附录里的信息只是些为了帮助你尽快入门而准备的速查资料，具体的示例和注意事项超出了本书的讨论范围，这方面的细节还请你自行参阅相关的应用软件或编程语言的文档。

A.1 grep

grep是一种用来对文件或标准输入文本进行文字搜索的Unix工具。根据你具体使用的命令选项，grep支持基本、扩展和Perl正则表达式。 104

- -E：使用扩展正则表达式。
- -G：使用基本正则表达式。
- -P：使用Perl正则表达式。





提示　你使用的命令选项不同，grep工具的功能和用途也就不同。大多数用户喜欢使用Perl正则表达式（见稍后的描述），因为这些是最标准的。

请注意以下事项：

- 在默认的情况下，grep将把包含着匹配的各个文本行全部显示出来；如果你只想查看匹配结果，请使用-o选项。
- 使用-v选项将对整个匹配操作进行求非——只显示不匹配的文本行。
- 使用-c选项将只显示匹配的总数而不是次匹配的细节。
- grep工具只能用来进行搜索操作，不能用来进行替换操作。换句话说，grep工具不支持替换功能。

A.2　JavaScript

JavaScript 1.x版本在String和RegEx对象的以下几个方法里实现了正则表达式处理。

- exec：一个用来搜索一个匹配的RegEx方法。
- match：一个用来匹配一个字符串的String方法。
- replace：一个用来完成替换操作的String方法。
- search：一个用来测试在某给定字符串里是否存在着一个匹配的String方法。

105

- split：一个用来把一个字符串拆分为多个子串的String方法。
- test：一个用来测试在某给定字符串里是否存在着一个匹配的RegEx方法。



注意　JavaScript 2里的正则表达式处理（Mozilla和另外几种比较新的浏览器可以支持）与JavaScript 1.x向后兼容并提供了更多的功能。

JavaScript对正则表达式的支持源自Perl语言，但需要注意以下几个问题：

- JavaScript使用命令行选项来管理全局的区分大小写搜索：g选项激活全局搜索功能，i选项让匹配操作不区分字母的大小写，这两个选项可以组合为gi。
- 其他命令行选项（版本4及以后的浏览器支持）包括：m，支持多行字符串；s，支持单行字符串；x，忽略正则表达式模式里的空白字符。
- 在使用回溯引用的时候，$`将返回被匹配字符串前面的所有东西，$`将返回被匹配字符串后面的所有东西，$+将返回最后一个被匹配的子表达式，$&将返回被匹配到的所有东西。
- JavaScript提供了一个名为RegExp的全局对象，在执行完一个正则表达式之后，你们可以通过这个对象获得与这次执行有关的信息。
- JavaScript不支持POSIX字符类。
- JavaScript不支持\A和\Z。

106

A.3　Macromedia ColdFusion

ColdFusion通过以下4个函数提供正则表达式支持。

- REFind()：执行搜索。
- REFindNoCase()：执行不区分字母大小写的搜索。
- REReplace()：执行替换。
- REReplaceNoCase()：执行不区分字母大小写的替换。



注意　ColdFusion还支持在<CFINPUT>标签里使用正则表达式进行输入检查的做法。不过，这个标签本身并不对正则表达式进行处理，它只负责把正则表达式传递到最终生成的客户端JavaScript代码里，对正则表达式的处理由JavaScript负责完成。因此，出现在<CFINPUT>标签里的正则表达式必须遵守JavaScript的有关正则和注意事项。

ColdFusion支持与Perl语言兼容的正则表达式，但需要注意以下几个问题：

- .总是匹配换行符。
- 在使用回溯引用的时候，必须使用\n代替$n来引用回溯引用变量。ColdFusion将自动地把替换字符串里的所有$字符解释为它们的转义含义。
- 你不必对替换字符串里的反斜线字符进行转义。ColdFusion将自动地把它们解释为它们的转义含义，但大小写转换序列或它们的转义版本（例如\u或\\u）属于例外。
- 嵌入限定符（(?i)、等）总是影响整个表达式，即使它们只是出现在某个子表达式的内部。

107

- ColdFusion 不支持在替换字符串里使用\Q、\u\L或\l\U。
- ColdFusion 不支持向后查找（?<=和?<!）。
- ColdFusion 不支持条件处理。
- ColdFusion 不支持\x、\N、\p和\C。



注意　在这本书里，我们提到ColdFusion的所有地方都指的是ColdFusion MX或更高版本。这些版本里的正则表达式引擎都经过了全面的改写，与早期版本里的大不一样；换句话说，本书没有对早期ColdFusion版本对正则表达式的支持进行讨论。

A.4　Macromedia Dreamweaver

Macromedia Dreamweaver支持在“搜索和替换”操作中使用正则表达式。

要想使用正则表达式，请按以下步骤操作。

- 在“Edit（编辑）”菜单里选择“Find and Replace（查找和替换）”，再选中“Use Regular Expression（使用正则表达式）”选择框。

请注意以下事项。

- 在替换模式里，必须使用$语法（例如$1）来引用一个回溯引用。但如果是在同一个模式里，则必须使用\语法（例如\1）来引用一个回溯引用。

- 正则表达式模式可以保存起来供以后再次使用。

A.5 Macromedia HomeSite（和ColdFusion Studio）

Macromedia HomeSite（包括ColdFusion Studio）支持在“搜索和替换”操作中使用正则表达式。 108

要想使用正则表达式，请按以下步骤操作：

- 在“Search（搜索）”菜单里选择“Extended Find（扩展查找）”或“Extended Repalce（扩展替换）”。
- 选中“Regular Expression（正则表达式）”选择框。

请注意以下事项：

- HomeSite正则表达式支持与ColdFusion里的正则表达式基本一致。
- HomeSite不支持POSIX字符类。
- HomeSite支持回溯引用，但使用的是`\1`语法。
- `.`总是匹配换行符。
- 正则表达式模式可以保存起来供以后再次使用。

A.6 Microsoft ASP

所有的ASP脚本语言都支持正则表达式。正则表达式支持是通过一个名为`RegExp`的对象提供的，这个对象包含着以下几个方法。

- `Excute()`：执行一个正则表达式搜索操作。
- `Replace()`：执行一个“搜索和替换”操作。
- `Test()`：检查一个字符串是否与一个给定的正则表达式相匹配。

ASP正则表达式支持还有一些局限性（ASP.NET有着非常高级和复杂的正则表达式支持）。下面是一些你们必须注意的事项。

- 在执行上述任何一个方法之前，必须先创建并填充一个`RegExp`对象的实例。
- 正则表达式被存放在`RegExp.Pattern`里。 109
- 支持全局限定符和大小写限定符。前者是一个存放在`RegExp.`

Global里的布尔值，后者是一个存放在RegExp.IgnoreCase里的布尔值。

- Excute()方法将返回一个Match对象，通过这个对象可以访问到所有的匹配。
- 不支持向前查找（?=和?!）和向后查找（?<=和?<!）。

A.7　Microsoft ASP.NET

ASP.NET里的正则表达式支持由.NET Framework提供。参阅后面的A.9节。

A.8　Microsoft C#

C#里的正则表达式支持由.NET Framework提供。参阅A.9节。

A.9　Microsoft .NET

.NET Framework通过它的基本类库提供了强大和灵活的正则表达式支持，这些支持在所有的.NET语言和工具（包括ASP.NET、C#和Visual Studio .NET在内）里都可以使用。

.NET里的正则表达式支持是通过Regex类（以及其他一些辅助类）提供的。Regex类有以下一些方法。

- IsMatch():测试在某个给定的字符串里是否可以找到一个匹配。
- Match()：搜索一个单个的匹配，该匹配将被为一个Match对象。
- Matches()：搜索所有的匹配，它们将被返回为一个Match-Collection对象。

110

- Replace()：在一个给定的字符串上进行替换操作。
- Split()：把一个字符串拆分为一个字符串数组。

利用各种包装器函数，在无须创建和使用一个Regex类的情况下也可以执行一个正则表达式。

- Regex.IsMatch()：在功能上等价于IsMatch()方法。
- Regex.Match()：在功能上等价于Match()方法。

- Regex.Matches()：在功能上等价于Matchex()方法。
- Regex.Replace()：在功能上等价于Replace()方法。
- Regex.Split()：在功能上等价于Split()方法。

下面是一些与.NET正则表达式支持有关的重要注意事项。

- 要想使用正则表达式，必须用"Imports System.Text.RegularExpressions"语句导入正则表达式对象。
- 如果只是临时需要使用正则表达式，上述包装器函数是理想的选择。
- 正则表达式的选项需要使用Regex.Options属性给出——它是一个RegexOption枚举集合，你可以对这个枚举集合的各有关成员如IgnoreCase、Multiline、Singleline等进行设置。
- .NET支持命名捕获，即允许对子表达式进行命名（这样就可以使用名字而不是编号来引用它们了）。命名一个子表达式的语法是?<name>，引用这个回溯引用的语法是\k<name>，在一个替换模式里引用它的语法是${name}。
- 在使用回溯引用的时候，$`（反引号）将返回被匹配字符串前面的所有东西，$'（单引号）将返回被匹配字符串后面的所有东西，$+将返回最后一个被匹配的子表达式，$_将返回整个原始字符串，$&将返回整个被匹配字符串。

111

- .NET Framework不支持使用\E、\l、\L、\u和\U进行大小写转换。
- .NET Framework不支持POSIX字符类。

A.10　Microsoft Visual Studio .NET

Visual Studio .NET里的正则表达式支持由.NET Framework提供。参阅前面的A.9节。

要想使用正则表达式，请按以下步骤操作：

- 在"Edit"菜单里选择"Find and Replace"。
- 选择"Find"、"Replace"、"Find in Files（在文件里查找）"或"Replace in Files（在文件里替换）"。

打开“Use（使用）”下拉框，从下拉清单里选择“Regular expressions”。

请注意以下事项：

- 使用@代替*?。
- 使用#代替+?。
- 使用^n代替{n}。
- 在替换操作里，可以用\(w, n)语法（其中的w是宽度，n是一个回溯引用编号）来左对齐一个回溯引用，右对齐一个回溯引用的语法是\(-w, n)。
- Visual Studio .NET使用以下特殊元字符和符号来表示常用的字符集合：

:a	[a-zA-Z0-9]
:c	[a-zA-Z]
:d	\d
:h	[a-fA-F0-9]（十六进制数字）
:i	[a-zA-Z_$][a-zA-Z_0-9$]*（合法的.NET标识符）
:q	一个括在引号里的字符串
:w	[a-zA-Z]+
:z	\d+

- \n是一个与平台无关的换行符。在替换操作里，它将插入一个新行。

112

- 支持以下几种特殊的字母匹配字符：

:Lu	匹配任意大写字母
:Ll	匹配任意小写字母
:Lt	匹配单词的标题形式（首字母是大写的单词）
:Lm	匹配任意标点符号

- 支持以下几种特殊的数字匹配字符：

:Nd	[0-9]+（十进制数字）
:Nl	罗马数字

- 支持以下几种特殊的标点符号匹配字符：

:Ps	配对标点符号的开始符号（左括号、左引号，等等）
:Pe	配对标点符号的结束符号（右括号、右引号，等等）
:Pi	双引号

:Pf 单引号
:Pd 短划线（连字符）
:Pc 下划线
:Po 其他标点符号

- 支持以下几种特殊的符号匹配字符：

:Sm 数学符号
:Sc 货币符号
:Sk 重音和方言符号
:So 其他符号

- 还有一些字符在.NET Framework里也有特殊含义，详细情况参阅Visual Studio .NET文档。

A.11 MySQL

MySQL是一个流行的开放源代码数据库软件。MySQL率先提供了正则表达式支持作为一种数据库搜索手段，这一点我们在其他数据库系统里还没有见过。

MySQL对正则表达式的支持体现在允许在WHERE子句里使用如下格式的表达式：

```
REGEXP "expression"
```



注意 下面是一条使用了正则表达式的MySQL语句的完整语法：

```
SELECT * FROM table WHERE REGEXP "pattern"
```

MySQL正则表达式支持很有用，功能也很强大，但它还算不上是一个完备的正则表达式实现。

- 只提供了搜索支持，不支持使用正则表达式进行替换操作。
- 在默认的情况下，正则表达式搜索不区分字母的大小写。如果需要区分字母的大小写，必须再增加一个BINARY关键字（放在REGEXP和模式之间）。
- 用[[:<:]]来匹配一个单词的开头，用[[:>:]]来匹配一个单词

118

的结束。

- 不支持向前预测。
- 不支持嵌入条件。
- 不支持八进制字符搜索。
- 不支持\a、\b、\e、\f和\v。
- 不支持回溯引用。

A.12　Perl

Perl可以说是各种正则表达式实现的“祖宗”，其他各种实现几乎都与Perl相兼容。

正则表达式支持是Perl的核心组件之一。如果需要在Perl脚本里使用正则表达式，只要像下面这样给出一个操作和相应的模式即可。

- `m/pattern/`　　匹配给定的模式。
- `s/ pattern/pattern/`　　执行一个替换操作。
- `qr/pattern/`　　返回一个Regex对象供今后使用。
- `split()`　　把一个字符串拆分为子字符串。

下面是一些与Perl正则表达式有关的注意事项。

- 允许把限定符放在模式的后面。\i用来表明在搜索时不区分字母的大小写；\g用来表明进行全局搜索（把所有的匹配都找出来）。
- 在使用“回溯引用”的时候，$\`将返回被匹配字符串前面的所有东西，$\`将返回被匹配字符串后面的所有东西，$+将返回最后一个被匹配的子表达式，$&将返回整个被匹配字符串。

114

A.13　PHP

PHP通过它的PCRE组件提供了与Perl相兼容的正则表达式支持。



注意　从PHP 4.2.0版本开始，PCRE组件将自动安装。PHP早期版本的用户需要自行编译PHP pcre-regex软件包才能启用正则表达式支持。

下面是PCRE组件提供的一些正则表达式函数。

- preg_grep()：进行一次搜索，匹配结果将作为数组返回。
- preg_match()：进行一次正则表达式搜索，返回第一个匹配。
- preg_match_all()：进行一次正则表达式搜索，返回所有的匹配。
- preg_quote()：这个函数的输入参数是一个模式，返回值是该模式的转义版本。
- preg_replace()：进行一次“搜索并替换”操作。
- preg_replace-callback()：进行一次“搜索并替换”操作，但使用一个回调（callback）函数来完成实际替换动作。
- preg_split()：把一个字符串拆分为子字符串。

请注意以下事项。

- 在默认的情况下，匹配操作不区分字母的大小写。如果不想区分字母的大小写，必须使用i限定符。
- 在默认的情况下，匹配操作仅限于单行字符串。如果需要匹配多行字符串，必须使用m限定符。
- preg_replace()、preg_replace_callback()和preg_split()函数都支持一个可选的参数，该参数用来给出一个上限值——对字符串进行替换或拆分的最大次数。 115
- preg_grep()和preg_replace_callback()是从PHP 4才开始有的，其他函数都是从PHP 3开始就被支持的。
- 在PHP 4.0.4和更高版本里，回溯引用可以用Perl语言的$语法（例如$1）来引用；在较早的版本里必须用\\来代替$。
- 不支持\l、\u、\L、\U、\Q和\v。

A.14 Sun Java

Java对正则表达式的支持是从1.4版本开始的，此前的JRE（Java Runtime Environment，Java运行环境）版本不支持正则表达式。



警告 版本低于1.4的JRE现在仍被广泛地使用着。如果你打算部署一个使用了正则表达式的Java应用程序，千万不要忘记检查JRE的版本。



注意 Sun公司花了很长的时间才在Java里实现了正则表达式支持，而不少软件团队在几年前就开发出了各种非官方的正则表达式实现。因为篇幅的限制，本书没能对那些非官方的Java正则表达式实现进行介绍；以下注意事项只适用于由Sun公司正式发布的Java正则表达式支持。

Java语言中的正则表达式匹配功能主要是通过java.util.regex.matcher类和以下这些方法实现的。

- `find()`：在一个字符串里寻找一个给定模式的匹配。
- `lookingAt()`：用一个给定的模式去尝试匹配一个字符串的开头。
- `matches()`：用一个给定的模式去尝试匹配一个完整的字符串。
- `replaceAll()`：进行替换操作，对所有的匹配都进行替换。
- `replaceFirst()`：进行替换操作，只对第一个匹配进行替换。

116

`matcher`类还提供了几个能够让程序员对特定操作做出更细致调控的方法。此外，`java.util.regex.pattern`类也提供了几个简单易用的包装器方法。

- `compile()`：把一个正则表达式编译成一个模式。
- `flags()`：返回某给定模式的匹配标志。
- `matches()`：在功能上等价于刚才介绍的`matches()`方法。
- `pattern()`：把一个模式还原为一个正则表达式。
- `split()`：把一个字符串拆分为子字符串。

Sun公司发布的Java正则表达式支持与Perl语言基本兼容，但要注意以下几点。

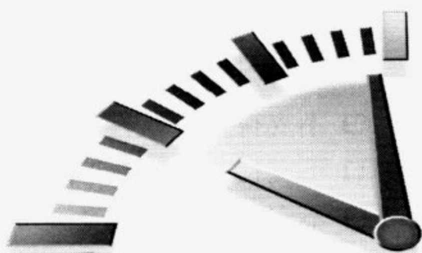
- 要想使用正则表达式，必须先用`import java.util.regex.*`语句导入正则表达式组件（这条语句将导入一个完整的软件包。

如果你只需要用到其中的一部分功能，请用相应的软件包名字替换掉这条语句里的*）。

- 不支持嵌入条件。
- 不支持使用\E、\l、\L、\u和\U进行字母大小写转换。
- 不支持使用\b匹配退格符。
- 不支持\z。

117





附录 B

常见问题的正则表达式解决方案

本附录收集了一些非常实用的正则表达式并对它们分别做了详细的解释。这些正则表达式所涉及的问题都是大家在实际工作中经常会遇到的。我们编写这个附录的目的有两个：一是通过解决这些实际问题对全书内容做一个总结，二是向大家提供一些现成的模式以帮助大家节省这方面的时间和精力。



注意 本附录里的示例不一定是相关问题的终极答案。事实上，正如我们在书中反复提到的那样，与正则表达式有关的问题很少会有一个终极的答案。更常见的情况是同时存在多种答案——它们没有绝对的对错之分，它们之间的区别只体现在你希望你的匹配操作严格到什么程度或者说你对匹配误差容忍到什么程度。在构造一个正则表达式模式的时候，我们不仅要考虑到匹配结果的准确性，还必须考虑到它的执行效率；而这两个因素往往难以两全。有了这样的认识，相信大家能够根据你们的具体情况选用这里给出的模式并在必要时对它们做出进一步改进。

B.1　北美电话号码[①]

North American Numbering Plan（北美编号方案）对北美地区的电话号码格式做出了定义。根据这一方案，北美地区（美国、加拿大、加勒比海地区大部以及其他几个地区）的电话号码由一个3位数的区号和一个7位数的号码构成（这7位数字又分成一个 3位数的局号和一个4位数的线路号，局号和线路号之间用连字符分隔）。每位电话号码可以是任意数字，但区号和局号的第一位数字不能是0或1。在书写电话号码的时候，人们往往把区号放在括号里，而且还往往会在区号与实际电话号码之间加上一个连字符来分隔它们。匹配(555) 555-5555（右括号的后面有一个空格）或(555)555-5555或555-555-5555其中之一很简单，但要想编写一个能够同时匹配这些电话号码的模式就不那么容易了。

118

文本

```
J. Doe: 248-555-1234
B. Smith: (313) 555-1234
A. Lee: (810)555-1234
```

正则表达式

```
\(?[2-9]\d\d\)?[ -]?[2-9]\d\d-\d{4}
```

结果

```
J. Doe: 248-555-1234
B. Smith: (313) 555-1234
A. Lee: (810)555-1234
```

分析

这个模式的开头是样子很怪的`\(?`，它负责匹配用来括住区号的括号——这对括号是可选的：`\(`匹配`(`字符，`?`表示匹配`(`的零次或一次出现。接下来的`[2-9]\d\d`负责匹配一个3位数的区号（第1位数字只能是2到

---

① **中国固定电话号码**
我国的固定电话号码的规律是，最开始的位一定是0，表示长途，接着是两位、三位或者四位数字组成的区号，然后是7位或者8位的电话号码，其中首位不为1（1用于特殊用途)。而国内习惯的电话格式有：029 8845 7890，029 88457890，(029) 8845 7890，(029) 88457890，029-8845 7890，029-88457890，029-8845-7890。对应的正则表达式可以写为：`\(?0[1-9]\d{1,3}\)?[-]?[2-9]\d{2,3}[-]?\d{4}`。——编者注

9）。\)?匹配一个可选的右括号，[-]?匹配一个空格或连字符——这个字符也是可选的。[2-9]\d\d-\d{4}匹配电话号码的剩余部分：一个3位数的局号（第1位数字只能是2到9）、一个连字符和最后4位数字。

只须稍做修改，这个模式就可以用来匹配北美电话号码的其他格式。比如像555.555.5555这样的号码。

119

文本

```
J. Doe: 248-555-1234
B. Smith: (313) 555-1234
A. Lee: (810)555-1234
M. Jones: 734.555.9999
```

正则表达式

```
[\(.]?[2-9]\d\d[\).]?[ -]?[2-9]\d\d[-.]\d{4}
```

结果

```
J. Doe: 248-555-1234
B. Smith: (313) 555-1234
A. Lee: (810)555-1234
M. Jones: 734.555.9999
```

分析

这个模式的开头部分使用了字符集合[\(.]?来匹配一个(或.字符——它们都是可选的。类似地，[\).]?匹配一个)或.字符——它们也都是可选的；[-.]匹配一个-或.字符。只要把这两个例子看明白了，你就可以轻而易举地把其他电话号码格式也添加到你的模式里。

B.2　美国邮政编码[1]

美国于1963年开始使用邮政编码（简称ZIP编码，ZIP是*Zone Improvement Plan*的首字母缩写）。美国目前有40 000多个ZIP编码，它们全都由数字构成（第1位数字代表从美国东部到西部的一个地域，0代表

① **中国邮政编码**

我国邮政编码的规则是，前两位表示省、市、自治区，第三位代表邮区，第四位代表县、市，最后两位代表投递邮局。共6位数字，其中第二位不为8（港澳前两位为99，其余省市为0-7）。对应的正则表达式可以写为：\d(9|[0-7])\d{4}。——编者注

东海岸地区，9代表西海岸地区）。在1983年，美国邮政总局开始使用扩展的ZIP编码，简称*ZIP+4*编码。新增加的4位数字对信件投送区域做了更细致的划分（细化到某个特定的城市街区或某幢特定的建筑物），这大大提高了信件的投送效率和准确性。不过，ZIP+4编码的使用是可选的，所以对ZIP编码进行检查通常必须同时照顾到5位数字的ZIP编码和9位数字的ZIP+4编码（ZIP+4编码中的后4位数字与前5位数字之间要用一个连字符隔开）。

文本

```
999 1st Avenue, Bigtown, NY, 11222
123 High Street, Any City, MI 48034-1234
```

正则表达式

```
\d{5}(-\d{4})?
```

120

结果

```
999 1st Avenue, Bigtown, NY, 11222
123 High Street, Any City, MI 48034-1234
```

分析

`\d{5}`匹配任意5位数字，`(-\d{4})?`匹配一个连字符和后4位数字。因为后4位数字是可选的，所以要把`-\d{4}`用括号括起来（这使它成为了一个子表达式），再用一个?来表明这个子表达式最多只允许出现一次。

B.3 加拿大邮政编码

加拿大邮政编码由6个交替出现的字母和数字字符构成。每个编码分成两部分：前3个字符用来给出FSA代码（*forward sortation area*，地区代码），后3个字符用来给出LDU代码（*local delivery unit*，街道代码）。FSA代码的第一个字符用来表明省、市或地区（这个字符有18种合法的选择；比如A代表纽芬兰地区；B代表新斯科舍地区；K、L、N和P代表安大略省；M代表多伦多市，等等），而我们的模式应该确保这第一个字符是合法的。在写出一个加拿大邮政编码的时候，FSA代码和LDU代码之间通常要用一个空格隔开。

文本

```
123 4th Street, Toronto, Ontario, M1A 1A1
567 8th Avenue, Montreal, Quebec, H9Z 9Z9
```

正则表达式

```
[ABCEGHJKLMNPRSTVXY]\d[A-Z] \d[A-Z]\d
```

结果

```
123 4th Street, Toronto, Ontario, M1A 1A1
567 8th Avenue, Montreal, Quebec, H9Z 9Z9
```

分析

[ABCEGHJKLMNPRSTVXY]匹配那18个合法字符中的任何一个，\d[A-Z]匹配一个数字和一个紧随其后的任意字母；它们合起来将匹配一个合法的FSA代码。\d[A-Z]\d匹配LDU代码，任意两个数字字符夹着任意一个字母。

121



注意 加拿大邮政编码不要求必须以大写形式写出，所以在使用上面这个正则表达式进行匹配时一般用不着区分字母的大小写。

B.4 英国邮政编码

英国邮政编码由5个、6个或7个字符构成，这些编码是由英国皇家邮政局定义的。英国邮政编码由两部分构成：代表邮政区划的外码(outcode)和代表城市街道的内码（incode）。外码是一个或两个字母后面跟着一位或两位数字，或者是一个或两个字母后面跟着一个数字和一个字母。内码永远是一位数字后面跟着两个字母（除C、I、K、O和V以外的任意字母——合法的英国邮政编码是不会在它的内码部分使用这5个字母的）。内码和外码之间要用一个空格隔开。

文本

```
171 Kyverdale Road, London N16 6PS
33 Main Street, Portsmouth, P01 3AX
18 High Street, London NW11 8AB
```

正则表达式

```
[A-Z]{1,2}\d[A-Z\d]? \d[ABD-HJLNP-UW-Z]{2}
```

结果

```
171 Kyverdale Road, London N16 6PS
33 Main Street, Portsmouth, P01 3AX
18 High Street, London NW11 8AB
```

分析

在这个模式里，[A-Z]{1, 2}\d匹配一个或两个字母紧跟着一位数字，随后的[A-Z\d]?匹配一个可选的字母或数字字符。于是，[A-Z]{1, 2}\d[A-Z\d]?将匹配任何一种合法的外码组合。内码部分由\d[ABD-HJLNP-UW-Z]{2}负责匹配，它将匹配任意一位数字和紧随其后的两个允许用在内码里的字母（A、B、D到H、J、L、N、P到U、W到Z）。 122



注意 英国邮政编码不要求必须以大写形式写出，所以在使用上面这个正则表达式进行匹配时一般用不着区分字母的大小写。

B.5 美国社会安全号码[①]

美国的社会安全号码（social security number，简称SSN号码）由3组以连字符隔开的数字构成：第1组包含着3位数字，第2组包含着2位数字，第3组包含着4位数字。从1972年起，美国政府开始根据SSN号码申请人提供的住址来分配第一组里的3位数字。

文本

```
John Smith: 123-45-6789
```

① **中华人民共和国公民身份号码**

可能是15位或者18位。前6位是户口所在地编码，其中第一位是1-8；此后是出生年月日，出生年份的前两位只能是18、19、20，而且是可选的（兼顾15位），月份中第一位只能是0或者1，日期的第一位只能是0-3；最后一位校验码是数字或者X，可选（兼顾15位）。对应的正则表达式可以写为：[1-8]\d{5}((18)|(19)|(20))?\d{2}[0-1]\d[0-3]\d{4}[\dx]?。——编者注

正则表达式

```
\d{3}-\d{2}-\d{4}
```

结果

```
John Smith: 123-45-6789
```

分析

`\d{3}-\d{2}-\d{4}`将依次匹配：任意3位数字、一个连字符、任意2位数字、一个连字符、任意4位数字。



注意 从理论上讲，SSN号码可以是任意数字组合，但从现实看，它们必须满足以下几项要求。首先，在一个合法的SSN号码里不可能出现全零字段；其次，第1组数字（到目前为止）不得大于728（因为SSN号码迄今为止还没用过那么大的数字，但未来可能会用到）。不过，一个能满足上述要求的模式会十分的复杂，因而比较简单的`\d{3}-\d{2}-\d{4}`更常见一些。

123

B.6 IP地址

IP地址由4个字节构成（这4个字节的取值范围都是0~255）。IP地址通常被写成4组以`.`字符隔开的整数（每个整数由1~3位数字构成）。

文本

```
localhost is 127.0.0.1.
```

正则表达式

```
(((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))\.){3}
➥((\d{1,2})|(1\d{2})|(2[0-4]\d)|(25[0-5]))
```

结果

```
localhost is 127.0.0.1.
```

分析

这个模式使用了一系列嵌套子表达式。我们先来说说由4个子表达式构成的`(((\d{1, 2})|(1\d{2})|(2[0-4]\d)|(25[0-5])\.)`：`(\d{1, 2})`匹配任意一位或两位数字（0~99）；`(1\d{2})`匹配以1开头

的任意三位数字（100~199）；(2[0-4]\d)匹配整数200~249；(25[0-5])匹配整数250~255。这几个子表达式通过|操作符结合为一个更大的子表达式（其含义是只须匹配这4个子表达式之一即可）。随后的\.用来匹配.字符，它与前4个子表达式构成的子表达式又构成了一个更大的子表达式，而接下来的{3}表明需要重复3次。最后，数值范围又重复了一次（这次省略了尾部的\.）以匹配IP地址里的最后一组数字。通过把4组以.分隔的数字的取值范围都限制在0~255之间，这个模式准确无误地做到了只匹配合法的IP地址，但不匹配非法的IP地址。



注意 第7章对这个IP地址的例子做了详细的解释。

124

B.7 URL地址

对URL地址进行匹配是一个有着相当难度的任务——其复杂性取决于你想获得多么精确的匹配结果。在最简单的情况下，你的URL匹配模式至少应该匹配到以下内容：协议名（http或https）、一个主机名、一个可选的端口号、一个文件路径。

文本

```
http://www.forta.com/blog
https://www.forta.com:80/blog/index.cfm
http://www.forta.com
http://ben:password@www.forta.com/
http://localhost/index.php?ab=1&c=2
http://localhost:8500/
```

正则表达式

```
https?://[-\w.]+(:\d+)?(/([\w/_.]*)?)?
```

结果

```
http://www.forta.com/blog
https://www.forta.com:80/blog/index.cfm
http://www.forta.com
http://ben:password@www.forta.com/
http://localhost/index.php?ab=1&c=2
http://localhost:8500/
```



分析

https?://匹配http://或https://（?使得字符s是可选的）。[-\w.]+匹配主机名。(:\d+)?匹配一个可选的端口号（参见上例中的第2和第6行）。(/([\w/_.]*)?)?负责匹配一个文件路径：外层的子表达式匹配一个可选的/字符，内层的子表达式匹配那个文件路径本身。正如大家看到的那样，这个模式不能正确处理查询字符串，也不能正确解读嵌在URL地址里的“username:password（用户名:口令字）”。不过，对绝大多数URL地址而言，这个模式的使用效果（匹配到主机名、端口号和文件路径）还是令人满意的。

125



注意　URL地址不要求必须以大写形式写出，所以在使用上面这个正则表达式进行匹配时一般用不着区分字母的大小写。



提示　如果你还想匹配使用了ftp协议的URL地址，把https?://替换为(http|https|ftp)即可。使用了其他协议的URL地址也可以按照类似的思路来匹配。

B.8　完整的URL地址

下面是一个更完备（也更慢）的URL地址匹配模式，它还可以匹配URL查询字符串（嵌在URL地址里的变量信息，这些信息与URL地址中的网址部分要用一个?隔开）以及可选的用户登录信息。

文本

```
http://www.forta.com/blog
https://www.forta.com:80/blog/index.cfm
http://www.forta.com
http://ben:password@www.forta.com/
http://localhost/index.php?ab=1&c=2
http://localhost:8500/
```



正则表达式

```
https?://(\w*:\w*@)?[-\w.]+(:\d+)?(/([\w/_.]*(\?\S+)?)?)?
```

结果

```
http://www.forta.com/blog
https://www.forta.com:80/blog/index.cfm
http://www.forta.com
http://ben:password@www.forta.com/
http://localhost/index.php?ab=1&c=2
http://localhost:8500/
```

分析

这个模式是在前一个例子的基础上改进而来的。这次紧跟在https?://后面的是(\w*:\w*@)?，它将匹配嵌在URL字符串里的用户名和口令字（用户名和口令字要用:隔开，它们的后面还跟着一个@字符），参见这个例子里的第4行。另外，这次在路径信息的后面还多了一个子表达式(\?\S+)?，它负责匹配查询字符串。查询字符串是在URL字符串里出现在?后面的文本，这些文本是可选的，所以这个子表达式的后面还紧跟着一个?。

126



注意　URL地址不要求必须以大写形式给出，所以在使用上面这个正则表达式进行匹配时一般用不着区分字母的大小写。



提示　能不能总是使用这个更完备的模式来取代前一个呢？从理论上讲，这没有什么不妥，但在实际工作中，因为这个模式比较复杂、处理速度也比较慢，所以如果没有特殊的必要，还是不使用它比较好。

B.9　电子邮件地址

用一个正则表达式来匹配电子邮件地址是一项很常见的任务。一般来说，如果需要匹配的电子邮件地址比较简单，相应的正则表达式就不会很复杂。

文本

```
My name is Ben Forta, and my
email address is ben@forta.com.
```

正则表达式

```
(\w+\.)*\w+@(\w+\.)+[A-Za-z]+
```

结果

```
My name is Ben Forta, and my
email address is ben@forta.com.
```

分析

(\w+\ .)*\w+负责匹配电子邮件地址里的用户名部分（@之前的所有文本）：(\w+\ .)*匹配一些由.结束的文本的零次或多次重复出现，\w+匹配必不可少的文本（这个组合将匹配ben和ben.forta，等等）。接下来，@匹配@字符本身，(\w+\ .)匹配至少一个以.结束的字符串，[A-Za-z]+匹配顶级域名（com、edu、us或uk，等等）。

合法的电子邮件地址必须在排版格式方面同时满足许多项规定。这个模式不能用来匹配每一种可能的电子邮件地址。比如说，这个模式会认为ben..forta@forta.com是一个合法匹配（但这显然不是一个合法的电子邮件地址），它不能用来匹配以IP地址做为主机名的电子邮件地址（但这种电子邮件地址是合法的）。不过，因为绝大多数电子邮件地址都能与这个模式相匹配，所以你不妨先用它试试，如果效果不佳再考虑对之进行改进。

127



注意　电子邮件地址不要求必须以大写形式写出，所以在使用上面这个正则表达式进行匹配时一般用不着区分字母的大小写。

B.10　HTML注释

HTML页面里的注释必须被放在<!--和-->标签之间（这两个标签必须至少包含两个连字符，多于两个没有关系）。在浏览（或调试）Web页面的时候，我们往往需要把所有的注释都找出来。

文本

```
<!-- Start of page -->
<HTML>
<!-- Start of head -->
<HEAD>
```

```
<TITLE>My Title</TITLE> <!-- Page title -->
</HEAD>
<!-- Body -->
<BODY>
```

正则表达式

```
<!-{2,}.*?-{2,}>
```

结果

```
<!-- Start of page -->
<HTML>
<!-- Start of head -->
<HEAD>
<TITLE>My Title</TITLE> <!-- Page title -->
</HEAD>
<!-- Body -->
<BODY>
```

分析

`<!-{2, }`匹配HTML注释的开始标签，也就是`<!`后面紧跟着两个或更多个连字符的情况。`.*?`匹配HTML注释的文字部分（注意，这里用的是一个懒惰型元字符）。`-{2, }>`匹配HTML注释的结束标签。

128



注意 这个模式匹配两个或更多个连字符，所以还可以用来查找CFML注释（这种注释的开始/结束标签里包含着3个连字符）。不过，这个模式没有对HTML注释的开始标签和结束标签所包含的连字符的个数是否配对进行检查（那可以用来检查HTML注释的格式是否有误）。

B.11 JavaScript注释

JavaScript（其他脚本语言如ActionScript和ECMA Script的其他变体等）代码里的注释都以`//`开头。正如刚才那个HTML注释的例子所示，把某给定页面里的所有注释全部查找出来是很有用的。

文本

```
<SCRIPT LANGUAGE="JavaScript">
// Turn off fields used only by replace
function hideReplaceFields() {
```

```
    document.getElementById('RegExReplace').disabled=true;
    document.getElementById('replaceheader').disabled=true;
  }
  // Turn on fields used only by replace
  function showReplaceFields() {
    document.getElementById('RegExReplace').disabled=false;
    document.getElementById('replaceheader').disabled=false;
  }
```

正则表达式

```
//.*
```

结果

```
<SCRIPT LANGUAGE="JavaScript">
// Turn off fields used only by replace
function hideReplaceFields() {
  document.getElementById('RegExReplace').disabled=true;
  document.getElementById('replaceheader').disabled=true;
}
// Turn on fields used only by replace
function showReplaceFields() {
  document.getElementById('RegExReplace').disabled=false;
  document.getElementById('replaceheader').disabled=false;
}
```

129

分析

这是一个很简单的模式：//.*匹配//和紧随其后的注释内容。



注意　与绝大多数正则表达式实现不同，在ColdFusion里，总是匹配换行符。因此，如果你正在使用的是ColdFusion，你将需要把这个模式修改成使用懒惰型元字符的样子，把.*替换为.*?。

B.12　信用卡号码

信用卡号码本身是否合法不能用正则表达式来检查，最终的结论要由信用卡的发行机构做出。我们这里说的检查是指使用正则表达式来检查信用卡号码的格式是否符合有关规定，其主要目的是为了——在对信用卡号码做进一步处理之前——把有打字错误的信用卡号码（比如多输入一位数字或少输入一位数字等情况）排除在外。



注意 这里使用的模式都有这样一个前提假设：信用卡号码里的空格和连字符已提前被去掉。一般来说，在使用正则表达式对信用卡号码进行匹配处理之前，先把其中的非数字字符去掉会给匹配操作带来很多便利；但这只是经验之谈，你应该根据具体情况来掌握。

所有的信用卡都遵守着同一种基本的编码模式——以特定的数字序列开头，号码的总位数是一个固定的值。我们先来看看MasterCard卡的情况。

130

文本

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 6011123456789012344
Diners Club: 38812345678901
```

正则表达式

```
5[1-5]\d{14}
```

结果

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 6011123456789012344
Diners Club: 38812345678901
```

分析

MasterCard卡的号码总长度是16位数字；第1位数字永远是5，第2位数字是1到5之一。`5[1-5]`匹配前两位数字；`{14}`匹配随后的14位数字。

Visa卡的情况稍微复杂一些。

文本

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
```

```
Amex: 371234567890123
Discover: 6011123456789012 34
Diners Club: 38812345678901
```

正则表达式

```
4\d{12}(\d{3})?
```

结果

131

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 6011123456789012 34
Diners Club: 38812345678901
```

分析

Visa卡的第1位号码永远是4，总长度是13或16位数字(不包括14或15，所以这里不能使用一个数字区间)。4匹配字符4本身，\d{12}匹配接下来的12位数字，(\d{3})?匹配可选的最后3位数字。

用来匹配美国运通卡号的模式相对要简单得多。

文本

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 6011123456789012 34
Diners Club: 38812345678901
```

正则表达式

```
3[47]\d{13}
```

结果

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 6011123456789012 34
Diners Club: 38812345678901
```

分析

美国运通卡的号码总长度是15位，前两位号码必须是34或37。3[47]匹配前两位数字，\d{13}匹配剩余的13位数字。

用来匹配Discover卡号的模式也很简单。

文本

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 601112345678901234
Diners Club: 38812345678901
```

132

正则表达式

```
6011\d{14}
```

结果

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 601112345678901234
Diners Club: 38812345678901
```

分析

Discover卡的号码总长度是16位，前4位号码必须是6011。6011\d{14}解决了问题。

Diners Club卡的情况稍微复杂一些。

文本

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 601112345678901234
Diners Club: 38812345678901
```

正则表达式

```
(30[0-5]|36\d|38\d)\d{11}
```

结果

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 601112345678901234
Diners Club: 38812345678901
```



分析

Diners Club卡的号码总长度是14位，必须以300到305、36或38开头。如果前三位号码是300到305，后面必须再有11位数字；如果前两位号码是36或38，则后面必须再有12位数字。我们这里采用了一个比较简单的办法：先匹配前3位数字——(30[0-5]|36\d|38\d)包含3个子表达式，只要其中之一得到匹配即可；其中30[0-5]匹配300~305，36\d匹配以36开头的任意3位数，38\d匹配以38开头的任意3位数。最后，\d{11}匹配剩余的11位数字。

133

现在，只要把上述5种信用卡号码的匹配模式组合成一个更大的模式就可以全面解决这个问题了：

文本

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 601112345678901234
Diners Club: 38812345678901
```

正则表达式

```
(5[1-5]\d{14})|(4\d{12}(\d{3})?)|(3[47]\d{13})|
➥(6011\d{14})|((30[0-5]|36\d|38\d)\d{11})
```

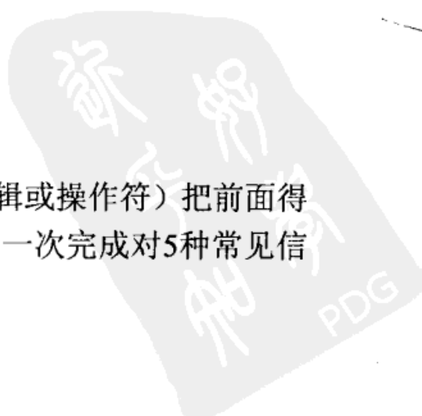
结果

```
MasterCard: 5212345678901234
Visa 1: 4123456789012
Visa 2: 4123456789012345
Amex: 371234567890123
Discover: 601112345678901234
Diners Club: 38812345678901
```

分析

这个模式用|操作符（正则表达式语言中的逻辑或操作符）把前面得到的5个模式结合到了一起。有了它，我们就可以一次完成对5种常见信用卡的号码格式进行检查了。

134





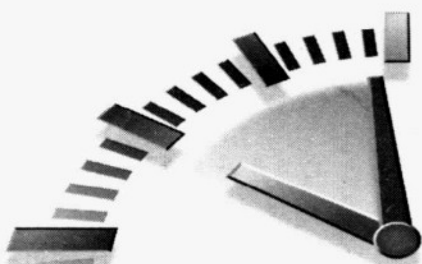
注意　这里使用的模式只能检查信用卡的号码是不是以正确的数字序列开头和是不是有着正确的总长度。不过，并非所有以4开头的13位数字都是合法的Visa卡号——信用卡号码还必须满足一个名为Mod 10的数学算法（这个算法适用于所有的信用卡类型）。在对信用卡进行编程处理的时候，Mod 10算法是一个必不可少的重要环节，但这项检查不属于正则表达式的工作——因为正则表达式不涉及数学运算。

B.13　小结

在本附录里，你看到了我们在前面的章程里介绍的许多概念和思路在实际工作中的应用例子。根据你遇到的具体问题，这些例子里的模式或者可以直接拿过来使用，或者需要稍做改动。我们把这些模式作为欢迎大家进入正则表达式世界的礼物，希望它们能帮助你拓展思路并在此基础上构造出更精彩更实用的模式来。

135





附录 C

正则表达式测试器

测试和试用正则表达式必须借助于一个应用软件或一种编程语言（很可能还需要编写一些代码）。为了帮助你学习和测试正则表达式，我们在本书的配套网站上提供了一个可以独立使用的Regular Expression Tester（正则表达式测试器）软件。本附录将简要地对这个软件的使用方法做一个介绍。

C.1 Regular Expression Tester软件

正则表达式不是一个应用程序，你不能通过点击某个链接的办法来使用它们。它们需要你编写正则表达式代码或是使用一个支持正则表达式的应用软件。

为了帮助你掌握正则表达式的用法，我们编写了一个简单的Web应用程序，它可以让你使用一个Web浏览器来测试和试用正则表达式。

这个应用程序只有一个页面，该页面包含着一个HTML表单和所有的正则表达式处理功能。你只要把有关的文件下载到你的计算机里就可以使用这个应用程序了。这个应用程序有许多种版本，其中包括一个对应于Microsoft ASP和ASP.NET以及一个对应于Macromedia ColdFusion的版本（它们都需要访问相应的服务器软件）和一个纯客户端的JavaScript版本。



提示 使用多个版本可以帮助大家测试某个模式的兼容性。



注意 我们将根据需要或读者的要求进一步提供针对其他平台的版本。如果你想获得一份这个测试器的副本或是希望了解都有哪些版本可供选择，访问我们在本附录的最后一节给出的URL地址。

C.1.1 进行查找操作

这个应用程序支持查找和替换操作。如果你想进行查找操作，请按以下步骤进行。

(1) 把这个应用程序的页面加载到你的浏览器里。
(2) 单击“Find”按钮。
(3) 在顶部的字段里输入你的正则表达式。
(4) 如有必要，选中“Case Sensitive”单选框。
(5) 在那个较大的文本框里输入（或通过剪贴操作）将被搜索的原始文本。
(6) 单击“Match Now”按钮。

查找结果将以一个表格的形式显示在这个表单的下部。

C.1.2 进行替换操作

替换操作需要用到两个模式。如果你想进行替换操作，请按以下步骤进行。

(1) 把这个应用程序的页面加载到你的浏览器里。
(2) 单击“Replace”按钮。
(3) 在顶部的字段里输入搜索正则表达式。
(4) 在第二个字段里输入替换正则表达式。
(5) 如有必要，请选中“Case Sensitive”单选框。
(6) 在那个较大的文本框里输入（或通过剪贴操作）将被搜索的原始文本。
(7) 单击“Match Now”按钮。

替换结果将显示在这个表单的下部。

C.2　获得这套应用程序的一份副本

要想获得这套应用程序的一份副本，请访问本书的配套网页。

http://www.forta.com/books/0672325667/

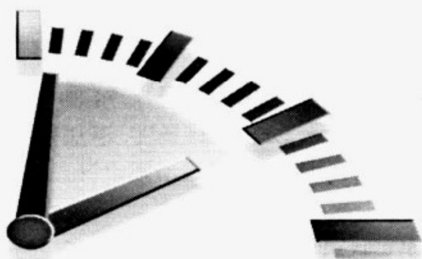
该页面的内容主要包括以下几项。

- Regular Expression Tester软件的现有版本和下载链接。
- 使用该软件的在线版本来测试各种正则表达式。
- 其他正则表达式资源的访问链接。
- 本书的勘误表（欢迎大家批评指正）。
- 如果你有兴趣把这个正则表达式测试器软件移植到其他平台上去的话，你还可以在那里找到申请加入相关团队的联系办法。

138

最后，欢迎大家进入正则表达式的精彩世界！





索 引

索引中页码为英文原书页码，与本书中页边标注的页码一致。

符号

D

E

N

O

P

V

W

X - Y - Z

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝正则表达式必知必会
作者＝（美）福达（Ｆｏｒｔａ，Ｂ．）著；杨涛等译
页数＝１３９
出版社＝北京市：人民邮电出版社
出版日期＝２００７．０７
ＳＳ号＝１１９１２３１６
ＤＸ号＝０００００６２５６１２７
ＵＲＬ＝ｈｔｔｐ：／／ｂｏｏｋ．ｓｚｄｎｅｔ．ｏｒｇ．ｃｎ／ｂｏｏｋＤｅｔａｉｌ．ｊｓｐ？ｄｘＮｕｍｂｅｒ＝０００００６２５６１２７＆ｄ＝１８５２Ｅ６８５９５７Ｄ９ＣＦＢ８１２１Ｅ８Ｄ５５Ｃ６Ｆ８Ｃ８Ｄ